UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Detección de patologías pulmonares en audios de pacientes

Autor: Miguel Arconada Manteca
Tutor: Manuel Antonio Sánchez-Montañés Isla

junio 2021

Miguel Arconada Manteca
Detección de patologías pulmonares en audios de pacientes

Miguel Arconada Manteca

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Las redes neuronales convolucionales (CNN) tienen numerosas aplicaciones en la clasificación y reconocimiento de imágenes. En este trabajo se emplean para otra finalidad, puesto que se busca clasificar audios de grabaciones de respiraciones de pacientes con diferentes diagnósticos de enfermedades respiratorias y pulmonares. De esta forma, se pretende ser útil en el diagnóstico precoz de enfermedades respiratorias.

Con el uso de herramientas como los espectrogramas *Mel* y el uso de *data augmentation*, se crea un algoritmo de preprocesamiento para poder preparar el conjunto de datos con el que se trabaja. Los audios analizados fueron introducidos en [1] en un trabajo conjunto de diversas universidades y hospitales de varios países.

Con el fin de implementar este proyecto, cabe destacar el uso de las librerías *librosa* para el procesamiento de los datos, y *TensorFlow* y *Keras* para el diseño de las redes convolucionales que aprenden sobre ellos.

Se han tenido en cuenta métricas adicionales a la tasa de acierto, como las denominadas *precision*, *recall* y *AUC*, que consideran conjuntos como los falsos positivos y negativos.

Se diseñan una serie de redes neuronales convolucionales con el fin de ir progresivamente mejorando la capacidad de aprendizaje sobre los datos disponibles.

Finalmente, se selecciona la que presenta mejores resultados sobre un conjunto de datos independiente al de entrenamiento. De esta forma, el modelo diseñado permite la posibilidad futura de añadir nuevos conjuntos de datos, siendo capaz de aprender sobre ellos y mejorar los resultados obtenidos.

Palabras clave

Aprendizaje automático, clasificación de audios, respiraciones, espectrogramas, *data augmentation*, red neuronal, convolución, *CNN*, *Keras*, *TensorFlow*

Abstract

Convolutional neural networks (*CNN*) have numerous aplications in image classification and recognition. In this study, they are used for another purpose, because the goal is audio classification of respirations of patients with diverse respiratory and lung diseases. This way, it is intended to be useful in the early diagnosis of respiratory diseases.

With the use of tools such as *Mel* spectrograms and *data augmentation*, a preprocessing algorithm is designed in order to be able to prepare the dataset we work with. The analyzed audio recordings were introduced in [1] in an study developed in several universities and hospitals from various countries.

In order to implement this project, it should be pointed out the use of libraries such as *librosa* for data processing, and *TensorFlow* and *Keras* for designing the convolutional neural networks which learn about them.

Additional metrics to the model accuracy, such as the *precision*, *recall* and *AUC*, have been taken into account. These consider quantities like false positives and negatives.

A series of convolutional neural networks are designed, with the aim of steadily increasing the ability to learn from available data.

Finally, the model that is selected is the one which presents the best results on a set of data independent from the learning dataset. That way, the designed model allows the future possibility of adding new data sets, being able to learn about them and to improve the results achieved.

Keywords

Machine learning, audio classification, respiratory sounds, spectrograms, *data augmentation*, neural network, convolution, *CNN*, *Keras*, *TensorFlow*

Índice

Listas

Lista de códigos

Lista de ecuaciones

Lista de figuras

1

Introducción

1.1. Motivación

Desde finales del año 2019, la humanidad está sufriendo la pandemia del denominado *SARS-CoV-2*, un virus que ha afectado a todos las sociedades de forma directa o indirecta. Los datos disponibles establecen en 170 millones el número de casos confirmados y en más de 5 millones las muertes. Además, las consecuencias económicas van a ser apreciables durante los próximos años en todos los países, después de entrar en recesión las principales economías del mundo.

En los últimos quince meses, numerosos han sido los estudios científicos dedicados a esta enfermedad, desde diversos puntos de vista y con diferentes objetivos. Así, los modelos matemáticos han sido de gran utilidad para la predicción de fases y tendencias en la infección, que permitían articular las normas de confinamiento. Por otra parte, un resultado directo del ingente esfuerzo científico y técnico realizado ha sido el desarrollo de una serie de vacunas en un tiempo menor del inicialmente esperado y del que fue necesario para las vacunas de otras enfermedades similares.

La relación entre la *Covid* y el *machine learning* ha sido una línea concreta de análisis e investigación con un importante desarrollo durante la pandemia. Estudios como [2] o [3] buscan aplicar esta disciplina a la detección de posibles brotes y focos de la infección, para poder mitigar los riesgos que esto conlleva. Otra aplicación se analiza en [4], donde se emplean técnicas de *deep learning* a la clasificación de radiografías de pacientes como método de detección precoz de complicaciones pulmonares de la enfermedad. Son diferentes enfoques técnicos para atender la necesidad global de erradicación de esta enfermedad, objetivo en el que la ingeniería informática ha mostrado aportaciones con una muy significativa utilidad.

Este estudio pretende profundizar sobre estas aplicaciones del aprendizaje automático a las consecuencias de esta infección, centrándose en la capacidad de detectar enfermedades respiratorias y pulmonares en grabaciones de audio de pacientes. De esta manera, se describe una herramienta que podría facilitar la detección de síntomas en la respiración compatibles con la *Covid*, haciendo posible una detección temprana, útil para que la intervención médica salve vidas o evite otras manifestaciones más graves de la enfermedad.

1.2. Objetivos

El objetivo principal de este trabajo es el diseño e implementación de una red neuronal que sea capaz de clasificar audios de forma satisfactoria, separando grabaciones de pacientes sanos de aquellas que indiquen la presencia de ciertas patologías respiratorias.

Para ello, será necesario procesar los datos disponibles para poder aplicarlos a nuestro modelo de aprendizaje sin perder aquellos contenidos que sean vitales para la clasificación de estos. Por lo tanto, el algoritmo de preprocesamiento deberá tener en cuenta las partes del audio que presentan la mayor información.

El modelo deberá responder correctamente a la adición de nuevos datos. Es decir, deberá ser capaz de generalizar su conocimiento sobre nuevas bases de datos que se le aporten. Además, es deseable que se puedan utilizar datos obtenidos siguiendo diversas metodologías, no únicamente audios obtenidos con instrumental profesional y en condiciones óptimas de grabación. El modelo deberá ser capaz de ignorar factores externos como ruido de fondo, problemas en las grabaciones, etc.

Por último, la red neuronal que se diseñe deberá tener un comportamiento fiable, desde el punto de vista de diferentes métricas. Será interesante analizar la tasa de acierto del algoritmo de clasificación desarrollado, pero también tener en cuenta otros aspectos como los falsos positivos o los falsos negativos, y las consecuencias que estos conllevan para posibles aplicaciones médicas.

2

Estado del arte

2.1. Aprendizaje automático en aplicaciones clínicas

En los últimos años ha habido una verdadera expansión de las aplicaciones del aprendizaje automático a la Medicina. El desarrollo de métodos de obtención de imágenes de diversas modalidades médicas, como pueden ser radiografías, resonancias o ecografías, ha ayudado a disponer de un mayor número de datos con los que entrenar diferentes redes con objetivos variados.

En [5] se realiza una revisión de los principales avances del campo de la Inteligencia Artificial en aplicaciones sanitarias en los últimos años. Entre otras, cabe destacar el uso de sensores implantables para el control de la vejiga, predicción de ataques epilépticos, derrames cerebrales, ...

El estudio [6] describe cómo, a partir de imágenes obtenidas por medio de resonancias magnéticas cerebrales, es posible diseñar algoritmos de detección automática de tumores. Pese a utilizar un conjunto de datos relativamente pequeño, mediante el uso de Perceptrón Multicapa y el algoritmo de *Naïve Bayes* es capaz de obtener tasas de acierto superiores al $90\,\%$.

Una aplicación similar se analiza en el estudio [7], donde se emplea el procesamiento de imágenes y el aprendizaje automático para detectar tumores mamarios en imágenes de mamografías. En concreto, se emplean dos tipos de redes, denominadas *Support Vector Machine* (SVM) y *Extreme Learning Machine* (ELM) . Esta aplicación permite un diagnóstico más eficiente y en etapas anteriores del desarrollo de uno de los cánceres que más víctimas mortales causa, según el estudio [8].

Como enunciaremos en la sección 2.3, un tipo de red neuronal denominado Red Neuronal Convolucional (CNN) ha sido ampliamente empleado para el análisis de imágenes y audios. En la sección 2.2 analizaremos con detenimiento el funcionamiento interno de estas redes.

En [9] se estudia el uso de CNNs, entre otros tipos de redes, para diseñar sistemas que analicen la dinámica de fluidos en el interior del corazón. Mediante estas técnicas, junto a la toma de imágenes médicas, se busca predecir el riesgo de trombos en el corazón detectando comportamientos extraños en el interior de este órgano.

Diversos estudios, como [10] o [11], analizan cómo el aprendizaje automático es capaz de realizar

tareas médicas como la detección de melanoma o la caracterización de componentes de material biológico, con unos resultados superiores a los obtenidos por profesionales sanitarios y científicos.

2.2. Redes convolucionales

Una CNN es un red neuronal profunda, es decir, que tiene múltiples capas ocultas entre la de entrada y salida. La característica que define esta subgrupo, y que las diferencia de otros tipos de redes neuronales, es el concepto de convolución. El uso de estas puede ser repetido en el interior de la red un número indeterminado de veces.

Una capa que realiza una convolución necesita realizar un producto de la entrada de la misma con una matriz o vector de menor dimensión, denominado "núcleo de la convolución", o en inglés *convolution kernel*. Usualmente, este tiene dimensiones menores que la entrada, luego es necesario realizar un producto Frobenius de las submatrices que la conforman, e ir desplazando el *kernel* sobre toda la matriz de la entrada.

Este producto se define, teniendo las matrices $A = (a_{ij})_{i\in I, j\in J}$ y $B = (b_{ij})_{i\in I, j\in J}$, de idénticas dimensiones, como se representa en las ecuaciones 2.1 y 2.2, siendo esta última la versión reducida.

$$\begin{aligned}\langle A, B\rangle_F = & A_{11}B_{11} + A_{12}B_{12} + \cdots A_{1n}B_{1n} \\ & + A_{21}B_{21} + A_{22}B_{22} + \cdots A_{2n}B_{2n} \\ & + \cdots \\ & + A_{m1}B_{m1} + A_{m2}B_{m2} + \cdots A_{mn}B_{mn}\end{aligned} \tag{2.1}$$

$$\langle A, B\rangle_F = \sum_{i,j} A_{ij}B_{ij} \tag{2.2}$$

Para poder realizar el producto sobre dos matrices de diferentes dimensiones, lo más habitual al no soler ser elevadas las dimensiones del *kernel*, hace falta definir cómo se multiplica por cada submatriz. Sea nuestra matriz de entrada A, con dimensiones $n \times m$, y K el *kernel* (K), con dimensiones $r \times s$. La matriz resultante Y tendrá dimensiones $(n-r+1) \times (m-s+1)$, y cada elemento se calcula como

$$Y_{ij} = \sum_{\tilde{i}=i-a}^{i+r-a} \sum_{\tilde{j}=j-b}^{j+s-b} A_{\tilde{i}\tilde{j}} K_{\tilde{i}-(i-a)\tilde{j}-(j-b)} \,. \tag{2.3}$$

De esta forma, mediante los parámetros a y b se define qué elementos de la matriz de entrada se utilizan para multiplicar por los elementos del *kernel* relativos a la posición de la matriz Y.

Por ejemplo, si tenemos las matrices

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{pmatrix}, \quad K = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.4}$$

obtenemos la matriz resultado

$$Y = \begin{pmatrix} 3 & 6 & 8 \\ 6 & 8 & 10 \end{pmatrix} \tag{2.5}$$

2.2.1. Capas habituales

Una vez definido el funcionamiento de una convolución, hace falta definir ciertas capas que encontraremos en las CNNs.

Capas convolucionales

Esta capa se encarga de realizar una convolución sobre su entrada y de pasar la salida como entrada a la siguiente capa.

Los datos recibidos definen un tensor con forma $(n\_inputs) \times (alto\_input) \times (ancho\_input) \times (canales\_input)$, y se transformarán en un *feature map* (mapa de características), ahora con dimensiones $(n\_inputs) \times (alto\_mapa) \times (ancho\_mapa) \times (canales\_mapa)$, donde cada parámetro representa:

- n_inputs: Número de datos de entrada que se le facilitan a la capa.
- $alto\_input$: Dimensión que representa el alto de cada entrada del conjunto.
- $ancho\_input$: Dimensión que representa el ancho de cada entrada del conjunto.
- $canales\_input$: Canales de cada entrada.
- $alto\_mapa$: Alto de la salida de la capa.
- $ancho\_mapa$: Ancho de la salida de la capa.
- $canales\_mapa$: Canales de la salida de la capa.

Los valores de alto y ancho de la salida serán menores que los de la entrada, como se puede observar en las ecuaciones 2.4 y 2.5. Será necesario definir el ancho y el alto del *kernel* de la convolución. En concreto, en la librería `keras` se puede realizar mediante el parámetro `kernel_size`. Tras esto, podemos obtener los valores

$$\begin{aligned} alto\_salida &= alto\_entrada - alto\_kernel + 1 \\ ancho\_salida &= ancho\_entrada - ancho\_kernel + 1 \end{aligned} \tag{2.6}$$

El valor de los canales de la salida será un parámetro introducido en el momento de definir la capa. En concreto, en la librería `keras` se denomina `filters`.

Capas *pooling*

Las CNN hacen uso de las capas denominadas *pooling*. Estas se encargan de reducir la dimensión de la capa anterior, combinando un cierto número de neuronas en una única neurona de salida. Podemos diferenciar dos grupos de opciones a la hora de definir una capa de esta familia: *global* o *local*, y *max* o *average*. Sea la entrada de la capa un tensor con forma $(n\_entradas, filas\_entrada, columnas\_entrada, canales\_entrada)$.

- *global*: La reducción se realiza mediante un cálculo sobre todas los valores de la entrada. Por tanto, la salida es un tensor de la forma $(n\_entrada, canales\_entrada)$.
- *local*: La reducción se realiza dividiendo la entrada en submatrices. Por tanto, es necesario definir el tamaño de la ventana de *pooling*. En la librería *keras* este argumento se llama *pool_size*. La salida tendrá forma $(n\_entradas, filas\_salida, columnas\_salida, canales\_entrada)$, donde los tamaños nuevos se calculan como

$$filas\_salida = \left\lfloor \frac{filas\_entrada}{filas\_pool\_window} \right\rfloor , \tag{2.7}$$

  y equivalentemente para las columnas.
- *max*: La operación que se calcula sobre los datos seleccionados, ya sean toda la entrada o un subconjunto es el máximo sobre este conjunto.
- *average*: La operación que se realiza sobre los datos es la media aritmética.

En la librería *keras* la familia de capas *pooling* está formada por las descritas a continuación, donde `{N}` representa la dimensionalidad de la operación, que puede tomar los valores 1, 2 o 3.

- `MaxPooling{N}D`,
- `AveragePooling{N}D`,
- `GlobalMaxPooling{N}D` y
- `GlobalAveragePooling{N}D`

Capas de normalización

A continuación, es necesario definir las capas que realizan normalización sobre su entrada. En la librería `keras` están definidas dos diferentes:

- `BatchNormalization`: Esta capa aplica una transformación a su entrada que busca mantener la media cercana a 0 y la desviación típica cerca de 1, sin modificar la forma del conjunto. La operación matemática, llamando *batch* al conjunto de entrada, devuelve para cada canal que se desee normalizar

$$\gamma \cdot \frac{batch - \overline{batch}}{\sqrt{\sigma^2(batch) + \varepsilon}} + \beta \tag{2.8}$$

  donde

- γ es un parámetro escogido al crear la capa que se modifica con cada entrenamiento.
- $\overline{batch}$ representa la media de los datos de entrada.
- $\sigma^2(batch)$ representa la desviación típica del conjunto de entrada.
- ε es un parámetro escogido para evitar la división entre 0.
- β es un parámetro escogido que se recalcula mediante el entrenamiento.

- `LayerNormalization`: Esta capa, a diferencia de la anterior, aplica la transformación a cada ejemplo en vez de a todo el conjunto de entrada simultáneamente.

Capas de *dropout*

Esta capa realiza una tranformación distinta a los datos que consiste, de forma aleatoria con una probabilidad configurable, en igualar a 0 una de las entradas. De forma que no se modifique la suma total, el resto se multiplican por $\frac{1}{1-p}$, donde p es la probabilidad escogida. De esta forma, esta capa ayuda a evitar el fenómeno conocido como *overfitting*, o sobreentrenamiento, que se da cuando el modelo empieza a perder capacidad de generalización sobre los datos de entrada, de forma que se obtienen peores resultados sobre el resto de datos.

Capas densas

Es la capa básica del modelo de redes neuronales. Toma la entrada y opera sobre las neuronas de la forma

$$y_j = f(\sum_i b_i \cdot x_i + b_0)\,, \tag{2.9}$$

donde cada parámetro representa:

- y_j representa la neurona de salida j.
- f es la función de activación que se aplica al resultado de la suma.
- b_i es el peso de la conexión de la neurona de entrada i a la neurona de salida.
- x_i es el valor de la neurona de entrada i.
- b_0 es conocido como el *bias*, o sesgo, un valor que se añade a la suma independientemente del valor de las neuronas.

2.2.2. Métricas de entrenamiento

Una vez podemos definir el modelo de red neuronal que queremos utilizar, debemos analizar las métricas que permitan evaluarlo, y poder inspeccionar la evolución del algoritmo de aprendizaje en cada época del mismo.

Accuracy

La métrica más habitual en los problemas de clasificación es la tasa de acierto, o *Accuracy*. Simplemente calcula la proporción de datos que son etiquetados correctamente del conjunto total. Por tanto, el cálculo es el descrito en la ecuación 2.10.

$$acc = \frac{predicciones\_correctas}{predicciones\_totales} \tag{2.10}$$

En el caso de tener datos correspondientes a las clases positivo y negativo, es interesante tener en cuenta no solo la tasa de acierto, sino cantidades como falsos positivos o positivos no diagnosticados. Por ello, se definen las siguientes métricas:

Precision

Esta métrica calcula la precisión de la predicción sobre los datos de *test*. Este valor es la proporción de datos correspondientes a la clase positivo etiquetados correctamente. Por tanto, una precisión será mejor cuanto más cercana a 1 sea, lo que significa un menor número de falsos positivos.

El cálculo es descrito en la ecuación 2.11, donde pr representa los datos etiquetados como positivos de forma correcta, y pf, los etiquetados de la misma manera erróneamente, es decir, los conocidos como falsos positivos.

$$p = \frac{pr}{pr + pf} \tag{2.11}$$

Recall

Esta métrica, similar a la anterior, calcula la razón entre el número de datos etiquetados de forma diferente. En este caso, la ecuación es la descrita en 2.12, donde nf representa los datos etiquetados de forma errónea como negativo, es decir, los positivos no diagnosticados. Un valor de *Recall* equivaldrá a un mejor funcionamiento cuanto más cercana a 1 sea, lo que corresponde a un menor número de positivos no diagnosticados.

$$r = \frac{pr}{pr + nf} \tag{2.12}$$

AUC

Es habitual hablar de la curva *ROC*, que tiene como objetivo representar la capacidad del modelo de detectar los casos positivos correctamente. Para ello, se representa en un gráfico el punto formado por la proporción de negativos etiquetados incorrectamente y la proporción de positivos etiquetados correctamente . Por tanto, una predicción óptima se corresponde al punto $(0, 1)$, puesto que no habrá negativos etiquetados incorrectamente y todos los positivos lo serán correctamente.

La curva se forma al tomar diversos puntos al ir modificando los subconjuntos de datos que se evalúan, o variando el valor del *threshold* que decide la clasificación entre las dos clases.

Por tanto, definimos la métrica *AUC*, acrónimo de *Area Under the Curve*, traducido como Área Bajo la Curva, como sugiere su propio nombre, tomando intervalos si se trata de un conjunto discreto de puntos. El valor estará comprendido entre 0 y 1, puesto que es el área del cuadrado definido en el gráfico. Un valor óptimo es de 1, puesto que en este caso los puntos se acercan al $(0, 1)$ y un valor de $0{,}5$ equivale a realizar la decisión aleatoriamente.

2.3. Análisis de audio

El aprendizaje automático tiene numerosas aplicaciones en el campo del análisis de audio.

En primer lugar, se pueden encontrar numerosos estudios que aplican CNNs a la clasificación de audio ambiente.

En [12] se describe un conjunto de arquitecturas de CNNs que son empleadas para detectar un total de 30871 diferentes etiquetas en un conjunto de más de 5 millones de horas de vídeos. Se emplean las arquitecturas `AlexNet` [13], `VGG` [14], `Inception` [15] y `ResNet` [16], que son redes neuronales creadas anteriormente para otras finalidades, y que se pueden reutilizar, para posteriormente comparar los resultados.

En [17] un equipo de investigadores desarrolló una CNN capaz de reconocer sonidos de diversos pájaros, en concreto satisfactorio en 43 especies diferentes. Como objetivo se toma ser capaz de detectar la presencia de ciertos especímenes que se desee controlar, por estar en estudio o en peligro de extinción. En este estudio se emplean tres formas de representar los audios: espectrogramas *Mel*, espectrogramas basados en componentes armónicos y espectrogramas basados en componentes percusivos. Estos permiten transformar el audio, que es una serie temporal de datos como amplitudes, a una imagen (o matriz) por medio de estudio de frecuencias como pueden ser las transformadas de *Fourier*.

El artículo [18] describe una nueva forma de tratar los datos en forma de audio para trabajar con ellos por medio de una CNN. Para evitar que esta se centre en el ruido ambiente, en vez de repre-

sentaciones en dos dimensiones como pueden ser los espectrogramas, diseñan una red neuronal tridimensional, que distinga entra las diferentes características de los audios.

En [19] se define una familia de CNNs que, junto a la transformación de audio a espectrogramas, permiten extraer características sobre el sonido ambiente. Como objetivo se enuncia facilitar esta tarea a dispositivos inteligentes. Una funcionalidad similar se describe en [20].

Este tipo de CNNs se aplica también para el análisis de grabaciones musicales. En [21] describen una implementación que permite clasificar grabaciones con un conjunto de múltiples etiquetas, de forma que una misma grabación puede tener diversas etiquetas. Para esta finalidad, se emplean las bases de datos MagnaTagATune [22], Million Song Dataset [23] y MTG-Jamendo [24].

Otra aplicación usual de las CNNs al análisis de audio es el reconocimiento de patrones en la voz, con diversas finalidades. En [25] emplean esta familia de redes para detectar emociones en grabaciones de voz. Se evalúan los resultados utilizando bases de datos externas, denominadas *Berlin EmoDB* [26] y *IEMOCAP* [27].

Una posterior aplicación de reconocimiento de patrones en la voz se describe en [28], donde se emplean CNNs junto a otras arquitecturas para detectar signos de Alzheimer en grabaciones de pacientes, lo que ayuda a la detección precoz de esta patología. La base de datos utilizada es la disponible para el *ADReSS challenge* [29].

Otra aplicación al análisis de grabaciones de la voz se desarrolla en [30], donde mediante redes neuronales basadas en las CNNs y espectrogramas *Mel* consiguen detectar rasgos de emociones en audios disponibles en las mismas bases de datos que utiliza [25], descritas en [26] y [27].

La mayor parte de estas aplicaciones utilizan la transformación de audio a imágenes mediante el concepto de espectrogramas. Estos son una representación visual de la evolución frente al tiempo de las frecuencias de un audio.

En primer lugar, el audio digital se particiona en ventanas, denominadas *chunks*, que pueden o no solaparse. Tras esto, se aplican transformadas de Fourier para calcular la intensidad de las frecuencias en esa ventana. El procesamiento de cada ventana se corresponde a calcular el cuadrado del módulo de la Transformada de Fourier de Tiempo Reducido (STFT) , como se enuncia en la ecuación 2.13, donde w es la ventana de frecuencias y t el instante de tiempo. Estos datos posteriormente se colocan de manera secuencial para formar la imagen final.

$$espectrograma(t, w) = |STFT(t, w)|^2 \tag{2.13}$$

Esta función, denominada STFT, realiza una transformación a la señal consistente en una multiplicación por una función ventana, que únicamente es distinta a 0 en una pequeña parte. De esta

forma, solamente depende el resultado de los instantes cercanos al punto donde se está calculando. Si tenemos nuestra señal temporal $x(t)$, se puede representar el cálculo como

$$STFT\{x(t)\}(\tau,\omega) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-i\omega t}\,dt \tag{2.14}$$

donde $w(\tau)$ es la función ventana, que usualmente es una ventana *Hann* o una ventana *Gauss* centradas en 0, el exponente ω la frecuencia. Por cómo está definida, esta función es la transformada de Fourier de la función compleja $x(t)w(\tau - t)$, que representa la evolución temporal de la señal frente al tiempo.

Esta operación se puede discretizar, en distintos *chunks*, que pueden solaparse para reducir cambios significativos en los puntos frontera entre ellos. Por tanto, en esta situación la señal pasa a ser $x[n]$, y la ventana $w[n]$, y debido a esto el cálculo se modifica y resulta

$$STFT\{x[n]\}(m,\omega) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-i\omega n} \tag{2.15}$$

Como hemos enunciado anteriormente, frecuentemente se utilizan los espectrogramas *Mel*, una variación de los cálculos anteriores, que se puede resumir en los siguientes pasos:

1.– Calcular la transformada de Fourier de la señal.

2.– Aplicar a los resultados la conocida como "escala Mel", definida como

$$m = 2595 \cdot \log_{10}\left(1 + \frac{f}{700}\right) = 1127 \cdot \ln\left(1 + \frac{f}{700}\right) \tag{2.16}$$

utilizando una ventana triangular o una ventana *Hann*.

3.– Tomar el logaritmo de las energías correspondientes a cada frecuencia de *Mel*.

4.– Tomar, simulando una señal formada por los anteriores valores, la transformada de Fourier conocida como Transformada de coseno discreta (DCT) , que se define como

$$F(n) = \frac{\sqrt{2}c(n)}{\sqrt{N}} \sum_{i=0}^{N-1} x(i)\cos\left[\frac{(2i+1)n\pi}{2N}\right] \tag{2.17}$$

donde $c(n) = \frac{1}{\sqrt{2}}$ para $n = 0$ y $c(n) = 1$ para el resto de valores, y N es la dimensionalidad de la señal $x(n)$.

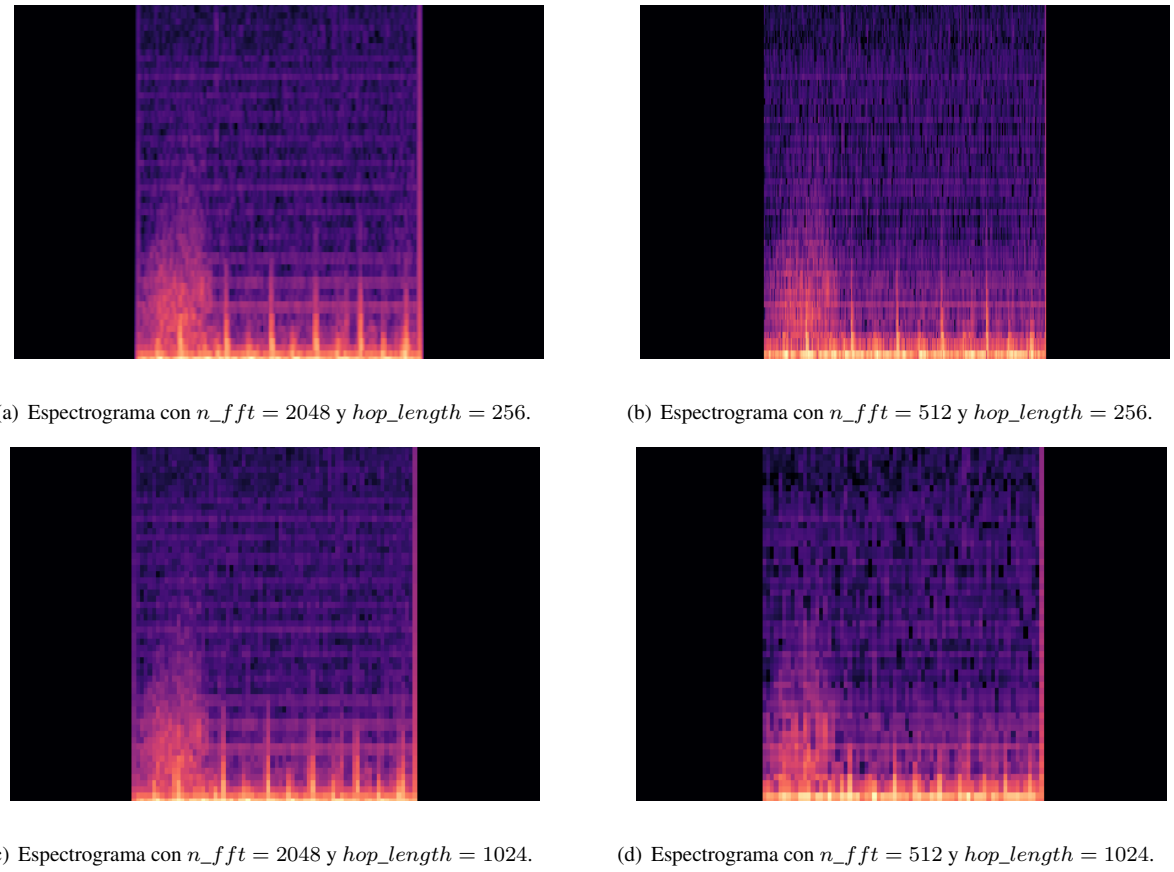
5.– Los coeficientes de Mel son los resultados del paso anterior.

Estas operaciones se describen con mayor nivel de detalle en [31] y [30].

Para nuestro estudio, podemos generar los espectrogramas *Mel* con el código descrito en 3.6, haciendo uso de la librería *keras*. En la línea 11 de este código utiliza los parámetros *hop_length*, que representa la longitud de la ventana, y `n_fft`, que representa el número de *samples* entre cada ventana.

Podemos representar diferentes espectrogramas correspondientes al mismo audio modificando estos dos parámetros como se puede observar en la figura 2.1. Podemos observar que un menor valor de `n_fft` implica un menor tamaño horizontal de las ventanas, lo que conlleva una mayor resolución pero

mayor diferencia entre cada una. Un mayor valor de `hop_length` resulta en una menor superposición de las diferentes ventanas, lo que resalta las diferencias entre las consecutivas, pero introduce mayor cantidad de ruido. Por ello, el valor de estos parámetros será una de las decisiones a estudiar y tener en cuenta a la hora de presentar los resultados.



(a) Espectrograma con $n\_fft = 2048$ y $hop\_length = 256$.

(b) Espectrograma con $n\_fft = 512$ y $hop\_length = 256$.

(c) Espectrograma con $n\_fft = 2048$ y $hop\_length = 1024$.

(d) Espectrograma con $n\_fft = 512$ y $hop\_length = 1024$.

Figura 2.1: Comparación de espectrogramas obtenidos modificando los parámetros `n_fft` y `hop_length`.

Por último, otro concepto a tener en cuenta a la hora de diseñar un algitmo de aprendizaje, y en concreto sobre el caso de audios, es el conocido como *data augmentation*. Esta funcionalidad permite obtener nuevos datos de entrenamiento, algo muy útil cuando el conjunto del que se dispone tiene un tamaño limitado. La librería *Keras* permite, haciendo uso de su clase *ImageDataGenerator*, realizar esta tarea.

En artículos como [32], [33], [34] o [35] se diseñan implementaciones que crean modificaciones de los audios originales. Entre las transformaciones se encuentra la conocida como *Pitch Shifting*, que aumenta o disminuye el timbre de una grabación de forma constante a lo largo de ella; *Time Stretching*, que se encarga de ralentizar o acelerar el audio sin modificar las frecuencias; o la introducción de diferentes sonidos, como puede ser ruido blanco o ruido ambiente. En [36] se introduce una forma de utilizar *data augmentation* mezclando diferentes audios del mismo conjunto de datos, así como alterar el orden dentro de un mismo audio.

3

Desarrollo

3.1. Descripción de los datos utilizados

El *dataset* utilizado para este trabajo fue introducido en [1]. Es el resultado de un trabajo conjunto de la Universidad de Coimbra (Portugal), la Universidad Aristóteles de Salónica (Grecia) y la Universidad de Aveiro (también de Portugal).

Este conjunto de datos contiene grabaciones realizadas por dos grupos de trabajo de forma independiente a lo largo de un periodo de varios años. En total, contiene más de 5 horas de audio, separadas en 6898 ficheros, cada uno de los cuales corresponde a un ciclo respiratorio independiente. Está formado por 920 grabaciones realizadas a 126 pacientes.

Los pacientes difieren en rasgos como edad, género o diagnósticos, aunque todos ellos presentaban grados variados de enfermedades respiratorias en el momento de la grabación.

El intrumental utilizado para las grabaciones es variado, e incluye un estetoscopio electrónico (Welch Allyn Meditron Master Elite Plus Stethoscope Model 5079-400), un fonendoscopio (3 M Littmann Classic II SE) y micrófonos eléctricos (AKG C 417 PP).

En el *dataset*, organizado en carpetas, podemos encontrar en primer lugar una serie de ficheros que describen el contenido:

En primer lugar, el fichero `demographic_info.txt` describe 6 cualidades de cada paciente:

- Número identificador del paciente.
- Edad.
- Sexo.
- Índice de masa corporal (BMI) , expresado en las unidades kg/m^2.
- Masa corporal, expresada en kilogramos.
- Altura, expresada en centímetros.

A pesar de no existir los seis datos para todos los pacientes, podemos extraer algunas características del grupo de pacientes: de los 126 pacientes, 46 son mujeres y 79 hombres. La edad media de

los pacientes es de 42 años, con una horquilla de edades desde 3 meses hasta 93 años.

A continuación, podemos encontrar el fichero `filename_format.txt`, que describe el convenio de nomenclatura para todos los ficheros de audio y texto. Estos serán de la siguiente forma:

```
<id-pac>_<id-grab>_<ubic>_<canal>_<instr>.<ext>
```

Cuadro 3.1: Formato de nomenclatura de los ficheros de audio y texto

En el esquema anterior, cada parte del nombre representa:

- `<id-pac>`: Código númerico de 3 dígitos que identifica de forma única a cada paciente.
- `<id-grab>`: Código alfanumérico que identifica de forma única a cada grabación.
- `<ubic>`: Código de dos caracteres que representa la ubicación del intrumental de grabación al hacer la grabación. Puede ser `Tc` para grabaciones en la tráquea, `A`, `P` o `L` seguido de `l` o `r` según sea una grabación en la parte anterior, posterior o lateral del pecho del paciente, en el lado izquierdo o derecho, respectivamente.
- `<canal>`: Código de dos caracteres que representa si la grabación se ha obtenido en modo de único canal (*single channel*, `sc`), o canal múltiple (*multi-channel*, `mc`).
- `<instr>`: Código alfanumérico que identifica el instrumental con el que se ha realizado la grabación. Las opciones posibles son `AKGC417L`, `LittC2SE`, `Litt3200` y `Meditron`.
- `<ext>`: Extensión del fichero. Es `wav` para los que contienen audio y `txt` para los de texto.

Cada fichero de audio contiene un ciclo respiratorio en formato `.wav`. Por otra parte, los ficheros de texto contienen una lista de líneas con cuatro columnas:

- Inicio de la respiración, medido en segundos desde el inicio del audio.
- Final de la respiración, medido en segundos desde el inicio del audio.
- Presencia de *crackles*. Un valor de `1` denota la presencia, y uno de `0` la ausencia.
- Presencia de *wheezes* (sibilancias). Un valor de `1` denota la presencia, y uno de `0` la ausencia.

Por último, podemos analizar el fichero llamado `patient_diagnosis.csv`, donde se describe el diagnóstico correspondiente a cada paciente. Por ello, es una tabla con únicamente dos columnas:

- Código identificador único del paciente.
- Diagnóstico correspondiente al paciente escrito en inglés. Los distintos valores son:
  - *URTI*: Se corresponde a la enfermedad denominada *Upper Respiratory Tract Infection* (URTI), es decir, Infección del tracto respiratorio superior, que comprende desde la cavidad nasal hasta la laringe.
  - *Healthy*: Representa las grabaciones de pacientes que no han sido diagnosticados ninguna de las enfermedades analizadas en este estudio.
  - *Asthma*: Se corresponde a la enfermedad denominada Asma: una enfermedad crónica que causa que las vías respiratorias de los pulmones se estrechen al hincharse.
  - *COPD*: Se corresponde a la enfermedad denominada *Chronic Obstructive Pulmonary Desease* (COPD), es decir, Enfermedad pulmonar obstructiva crónica.
  - *LRTI*: Se corresponde a la enfermedad denominada *Lower Respiratory Tract Infection* (LRTI), es decir, Infección del tracto respiratorio inferior, que comprende desde la tráquea a los bronquios.

- *Bronchiectasis*: Se corresponde a la enfermedad denominada Bronquiectasia, que ocasiona un ensanchamiento permanente de las vías respiratorias.
- *Pneumonia*: Se corresponde a la enfermedad denominada Neumonía, infección que causa que los alveolos pulmonares se llenen con líquido a pus.
- *Bronchiolitis*: Se corresponde a la enfermedad denominada bronquiolitis, inflamación en los bronquiolos, que se llenan de mucosidad y dificulta la respiración.

De este grupo de ficheros podemos extraer ciertas características estadísticas sobre el total de datos, que posteriormente en la sección 3.2 serán analizadas y discutidas para escoger el conjunto de datos de trabajo.

En primer lugar, se puede analizar la proporción de audios correspondientes a cada diagnóstico. En la figura 3.1 se puede observar que la gran mayoría (en concreto un $83{,}29\,\%$ del total de los datos) se corresponden con el diagnóstico COPD, mientras que los restantes representan una fracción despreciable del total.

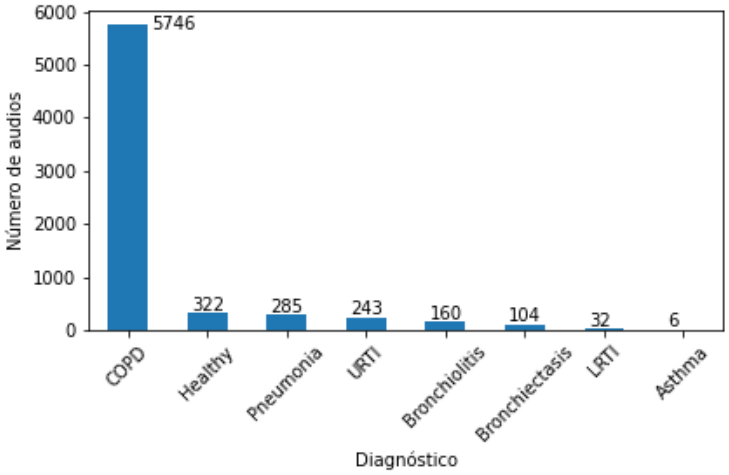


Figura 3.1: Reparto de audios según el diagnóstico del paciente

A continuación, la figura 3.2 permite visualizar la proporción de audios extraídos con cada instrumento de grabación. La mayor parte (que representa un $63{,}00\,\%$ del total) se ha realizado con el instrumento etiquetado como `AKGC417L`, que se corresponde al micrófono electrónico *AKG C 417 PP*. Como se verá en la sección 3.2, esta diferencia será importante a la hora de seleccionar los datos con los que trabajar.

La figura 3.3 representa gráfica la proporción de audios tomados en modo de canal múltiple o simple. Esta separación es una consecuencia directa del instrumental empleado, pues es el que de forma última selecciona el modo de grabación.

Por último, la figura 3.4 permite visualizar la localización del instrumental de grabación en el momento de la grabación. Los valores más frecuentes corresponden a posiciones en la parte anterior del pecho del paciente, frente a los valores menos frecuentes, que a su vez se corresponden a una

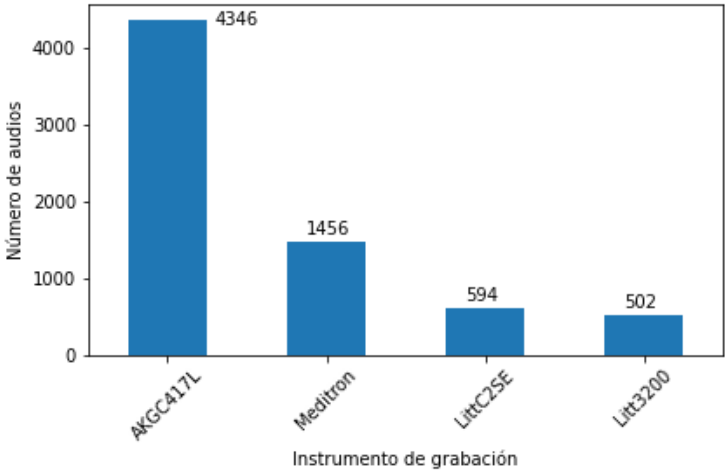


Figura 3.2: Reparto de audios según el instrumento de medida empleado en la grabación

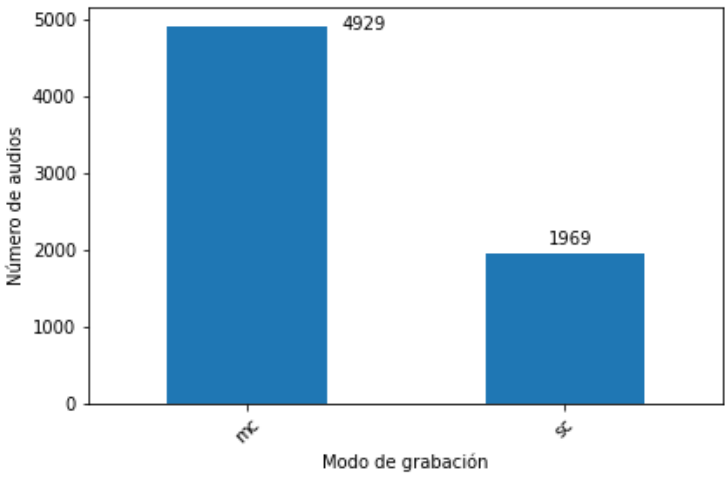


Figura 3.3: Reparto de audios según el modo de grabación

colocación en el lateral del pecho del mismo.

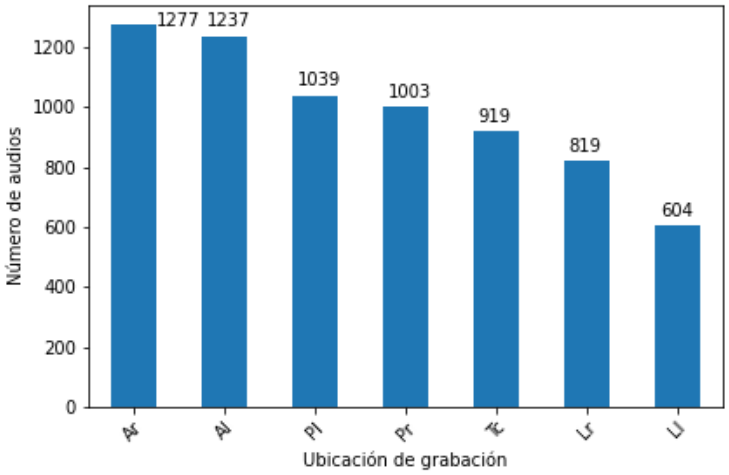


Figura 3.4: Reparto de audios según la ubicación del instrumental de medida al realizar la grabación

3.2. Preprocesamiento de los datos utilizados

3.2.1. Selección de los datos a usar

Una vez examinadas las características de los datos disponibles, hay que realizar una selección de los que se van a utilizar, y prepararlos para posteriormente operar sobre ellos.

En primer lugar, teniendo en cuenta la gran diferencia que se puede apreciar en la figura 3.2, y buscando no utilizar audios obtenidos mediante diferente instrumentos de grabación, nos decantamos por seleccionar los datos etiquetados como `AKGC417L`, ya que esto permitiría disponer de un total de 4346 de las 6898 respiraciones.

Sin embargo, esto presentan un grave problema, pues la totalidad de los datos obtenidos mediante este instrumento se corresponden al diagnóstico `COPD`, con lo cual pierde el sentido clasificar datos que pertenecen todos a una misma clase.

Por ello, nos vemos obligados a utilizar otro instrumento de medida, que permita utilizar datos de la clase *Healthy*, ya que como mínimo deseamos separar entre respiraciones de pacientes sanos y con alguna afección respiratoria. Agrupando las respiraciones disponibles según el diagnóstico, aquellas que corresponden a la clase deseada son todas realizadas con el instrumento `Meditron`, que se corresponde al estetoscopio electrónico *Welch Allyn Meditron Master Elite Plus Stethoscope Model 5079-400*. Debido a esta limitación, únicamente podemos emplear 1456 respiraciones, que representa un $21{,}10\,\%$ del total de 6989 audios.

La distribución del número de audios correspondientes a cada diagnóstico habrá variado si nos

limitamos a aquellos que hayan sido etiquetados con `Meditron`, como podemos observar en la figura 3.5. El diagnóstico `COPD` permanece el más frecuente, pero la diferencia se ha minimizado, al limitarnos a aquellos datos que han sido obtenidos con el instrumento etiquetado como `Meditron`. Además, ha desaparecido la clase `Asthma`, pues todos los audios correspondientes fueron tomados con otro instrumental.

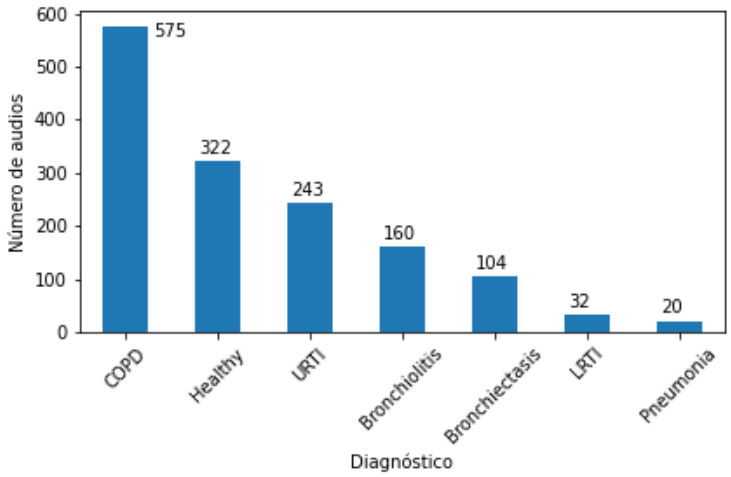


Figura 3.5: Reparto de audios según el diagnóstico del paciente para aquellos obtenidos con el `Meditron`.

La siguiente decisión que tomamos es separar solamente en dos clases correspondientes a paciente sano y paciente con cualquier afección. El reparto de audios por clase pasa a ser el representado en la figura 3.6. La diferencia entre las dos clases será un problema tratado más adelante, puesto que la clase *Healthy* representa un $22{,}12\,\%$ del total, lo que crea un *bias* a la hora de aprender sobre estos datos.

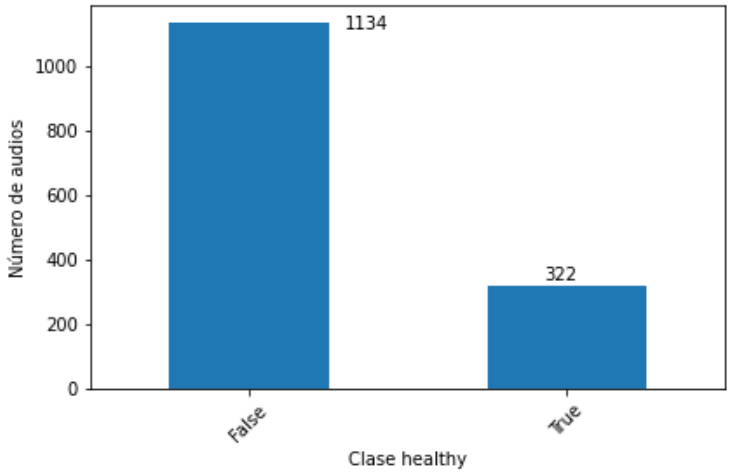


Figura 3.6: Reparto de audios según la clase para aquellos obtenidos con el `Meditron`.

3.2.2. Selección de la longitud de audios

Una vez definidos los datos con los que trabajar, y separados en las dos clases a clasificar, el siguiente paso es extraer de cada fichero de audio cada una de las respiraciones etiquetadas, ya que como comentamos anteriormente múltiples ciclos respiratorios se agrupan en cada uno.

En los correspondientes ficheros de texto, como describimos anteriormente, las dos primeras columnas enuncian el inicio y el final de cada respiración. Utilizando estos datos, podemos calcular la longitud de lo que llamamos los *slices* de cada audio. Para ello, en primer lugar leemos los datos a un *dataframe*, que permite representarlos y referenciarlos como una tabla. Por ejemplo, las primeras líneas son representadas en la tabla 3.1. Se puede observar que las diferentes entradas corresponden a respiraciones secuenciales, puesto que el tiempo de final de cada una es el de inicio de la siguiente.

start	end	filename	pId	ac_mode	instrument	diagnosis
1.862	5.718	160_1b3_Al_mc_AKGC417L	160.0	mc	AKGC417L	COPD
5.718	9.725	160_1b3_Al_mc_AKGC417L	160.0	mc	AKGC417L	COPD
9.725	13.614	160_1b3_Al_mc_AKGC417L	160.0	mc	AKGC417L	COPD
13.614	17.671	160_1b3_Al_mc_AKGC417L	160.0	mc	AKGC417L	COPD
17.671	19.541	160_1b3_Al_mc_AKGC417L	160.0	mc	AKGC417L	COPD

Tabla 3.1: Primeras líneas de la tabla que contiene los detalles de cada respiración

Por lo tanto, la longitud de cada respiración etiquetada se puede calcular como se representa en el código 3.1, y obtenemos que es de más de 16 segundos.

Código 3.1: Código para la visualización de la longitud máxima de los *slices*.

```
df_ficheros_escogidos['longitud_slice'] = df_ficheros_escogidos['end'].sub(df_ficheros_escogidos['start'],
    axis = 0)
print(max(df_ficheros_escogidos['longitud_slice']))
```

Sin embargo, esto por sí solo no es suficiente para determinar el valor a escoger. Para ello, representamos gráficamente las distintas longitudes de los *slices*, como se puede ver en las figuras 3.7 y 3.8.

Debemos tener en cuenta estos gráficos, y especialmente la figura 3.8, que representa una caja que extiende del primer al tercer cuartil, y segmentos que abarcan todos los datos excepto los denominados como *flier points*, que están suficientemente separados de los demás. Podemos observar que el valor de 16 segundos es algo aislado y nada frecuente. Haciendo uso de este tipo de gráfico, que analiza propiedades aleatorias, podemos ejecutar la línea de código descrita en 3.2, que toma el techo del valor del segmento superior, y obtenemos una duración deseada de 6 segundos.

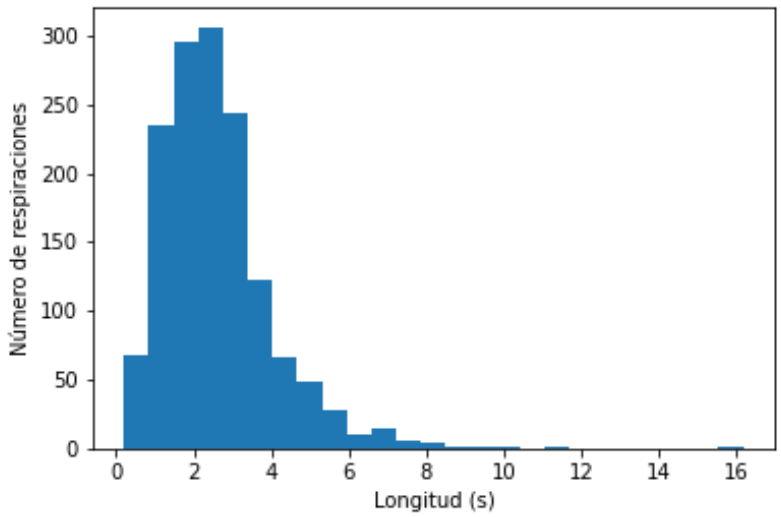


Figura 3.7: Valores de la longitud de cada *slice* representados en forma de histograma.

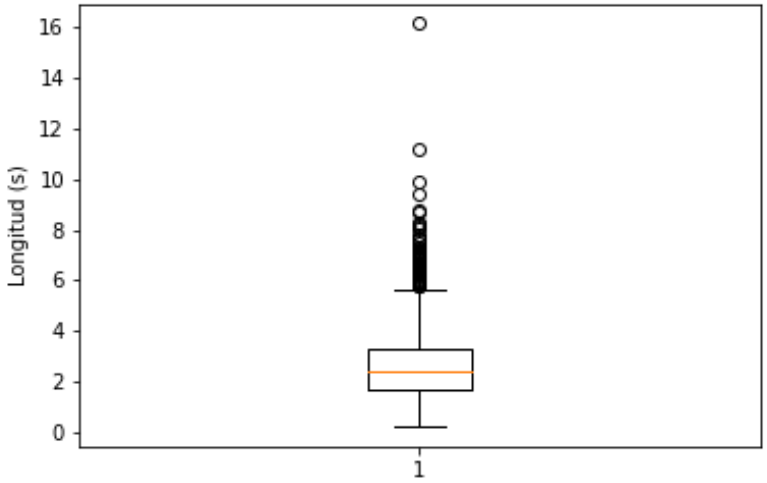


Figura 3.8: Valores de la longitud de cada *slice* representados en forma de *boxplot*.

Código 3.2: Código para el cálculo de la longitud deseada de los *slices*.

```
longitud_maxima = math.ceil(boxplot_stats(df_ficheros_escogidos['longitud_slice'])[0]['whishi'])
```

3.2.3. Extracción de respiraciones individuales

Una vez escogido el valor de 6 segundos como duración objetivo para todas las grabaciones, dado que ninguna tiene exactamente esa duración, necesitamos modificar los audios. Para ello, en primer lugar definimos la función auxiliar que se puede observar en el código 3.3. Esta se encarga de, dado una longitud en segundos, una frecuencia de muestreo y un modo de grabación calcular la duración objetivo del *array* que describe al audio.

Código 3.3: Código que calcula la longitud de los audios medida en muestras.

```
# Supuesta longitud de los datos del slice
def calcular_longitud(sample_rate=22050, time=longitud_maxima, modo='mc'):
    if modo == 'sc': # Canal único representa 'mono'
        return sample_rate *time
    else: # Múltiple canal representa 'stereo'
        return (sample_rate *time) *2
```

A continuación, extraemos de cada fichero de audio cada una de las respiraciones de forma independiente, y las organizamos en carpetas según el diagnóstico del paciente, teniendo en cuenta que sólo deseamos clasificar entre la clase correspondiente a pacientes sanos y aquellos con alguna afección. Para ello, empleamos la función descrita en el código 3.4, que toma los datos *raw* de un audio y sabiendo la posición de inicio y final de una respiración crea un recorte de los mismos. A continuación, esta función es aplicada en el código descrito en 3.5, en el que se leen los datos de las diferentes grabaciones, que a su vez se obtienen haciendo uso de la librería `librosa`. Tras separar cada respiración, estas se limitan a los 6 segundos escogidos o se rellena simétricamente con silencio si no se llega a esta longitud, para posteriormente crear un nuevo fichero de audio en la carpeta correspondiente, mediante la función `write` de la librería `soundfile`, importada como `sf`.

Código 3.4: Código para el recorte de un audio.

```
def slice_data(start, end, raw_data, sample_rate, recortado=False):
    max_ind = len(raw_data)
    start_ind = min(int(start *sample_rate), max_ind)
    end_ind = min(int(end *sample_rate), max_ind)
    if recortado:
      end_ind-=1
    return raw_data[start_ind: end_ind]
```

Código 3.5: Código para la extracción de los *slices* de los ficheros de audio y su organización en carpetas.

```
foo = 0
for id_fila, fila in df_ficheros_escogidos.iterrows():
    nombre_fichero = fila['filename']
    inicio = fila['start']
    final = fila['end']
    diagnostico = fila['diagnosis']

    if (final-inicio) > longitud_maxima:
        recortado = True
        final = inicio + longitud_maxima
    else:
        recortado = False
    audio_completo = audios_path + '/' + nombre_fichero + '.wav'
    if id_fila != 0:
        if df_ficheros_escogidos.iloc[id_fila-1]['filename'] == nombre_fichero:
            foo+=1
        else:
            foo=0

    data, sampling_rate = lb.load(audio_completo, sr=None, mono=False)
    sliced_data = slice_data(start=inicio, end=final, raw_data=data, sample_rate=sampling_rate,
            recortado=recortado)
    longitud_audio = calcular_longitud(sample_rate=sampling_rate, modo=fila['ac_mode'])
    data_con_padding = lb.util.pad_center(sliced_data, longitud_audio)

    nombre_fichero_ = nombre_fichero + '_' + str(foo) + '.wav'
    path = 'output/audio/' + [diagnostico if diagnostico == 'Healthy' else 'Unhealthy'] + '/' +
            nombre_fichero_
    sf.write(file=path, data=data_con_padding, samplerate=sampling_rate)
```

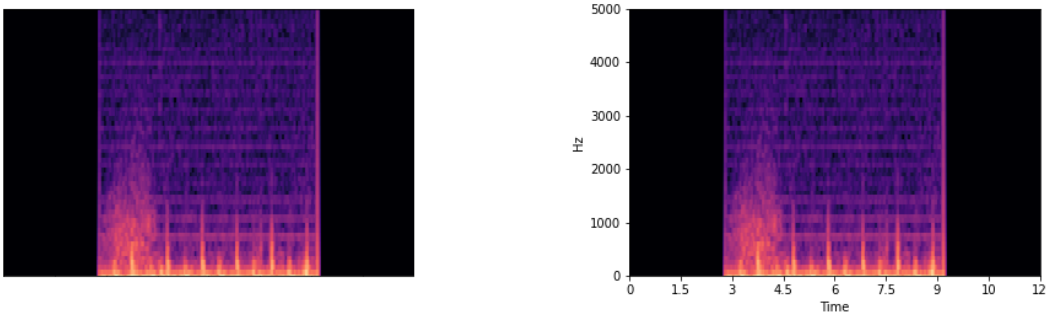
3.3. Aprendizaje sobre los datos utilizados

Como citaba en el capítulo 2, numerosas son las aplicaciones de los espectrogramas al reconocimiento de audio y, en concreto, a patrones de voz. Por ello, el primer paso de nuestro algoritmo de aprendizaje es generarlos a partir de los audios seleccionados en el paso anterior. Para ello, en primer lugar definimos la función `convertir_a_spectrograma`, que toma como entrada diversos parámetros que definen la ruta del fichero de audio, así como la clase a la que pertenece, y se encarga de generar el espectrograma correspondiente. En la primera versión, a la que corresponde este bloque de código posteriormente la imagen se guarda en ficheros organizados en carpetas. Sin embargo, en la sección 4 describimos otras versiones de esta función y su comportamiento tras las modificaciones.

El código que conforma esta función puede analizarse en 3.6. Es importante hacer notar que de nuevo utilizamos la librería `librosa` para leer el audio desde el fichero, y para calcular los coeficientes de *Mel* correspondientes al audio, ya que simplifica los cálculos de estos al limitarse a invocar a la función `librosa.feature.melspectrogram()`.

Los parámetros *n_fft*$= 2048$, *hop_length*$= 512$, que representan la longitud de la ventana *FFT*, y el número de *samples* entre frames consecutivos, respectivamente, han sido escogidos teniendo en cuenta qué pareja proporcionaba la mejor visualización de los espectrogramas, sin introducir demasiado ruido adicional a los mismos.

Las líneas 17 hasta la 21 del bloque de código 3.6 permiten ocultar los ejes de la gráfica resultante al llamar a la función `specshow` de la librería `librosa`, así como las etiquetas de los mismos, para poder obtener únicamente la imagen del histograma. En la figura 3.9 podemos observar la diferencia que consiguen estas líneas del código sobre la imagen obtenida.



(a) Espectrograma con los ejes ocultos.

(b) Mismo espectrograma sin ocultar los ejes.

Figura 3.9: Comparación de espectrogramas obtenidos mediante la función descrita en 3.6 y sin ocultar los ejes.

En la figura 3.10 podemos observar el espectrograma correspondiente a un audio, en concreto el llamado `119_1b1_Ar_sc_Meditron_5.wav`, que corresponde a un paciente diagnosticado con la

Código 3.6: Código de generación del espectrograma correspondiente a un fichero de audio y su archivo como imagen.

```
def convertir_a_spectrograma(path_ficheros, fichero, categoria, is_train=False):

    train_ = 'train/'
    test_ = 'test/'

    path_completa = path_ficheros + (train_ if is_train else test_) + categoria + '/' + fichero

    y, sr = lb.load(path_completa)

    plt.figure(figsize=(5,3))
    mel_spect = lb.feature.melspectrogram(y=y, sr=sr, n_fft=2048, hop_length=512)
    mel_spect = lb.power_to_db(mel_spect, ref=np.max)
    src_db=mel_spect

    specshow(src_db, sr=sr, x_axis='time', y_axis='hz')

    plt.gca().set_axis_off()
    plt.subplots_adjust(top = 1, bottom = 0, right = 1, left = 0, hspace = 0, wspace = 0)
    plt.margins(0,0)
    plt.gca().xaxis.set_major_locator(plt.NullLocator())
    plt.gca().yaxis.set_major_locator(plt.NullLocator())

    save_carpeta = 'output/images/'
    fichero_img = fichero.split('.wav')[0]

    save_path = path_root + '/' + save_carpeta + (train_ if is_train else test_) + categoria + '/' +
        fichero_img + '.png'

    plt.savefig(save_path, dpi=100, facecolor="none", bbox_inches = 'tight', pad_inches = 0)
    plt.close()
```

patología *URTI*. Podemos claramente observar que esta respiración tenía una longitud menor que la decidida anteriormente (6 segundos), por lo que presenta silencio al inicio y al final.

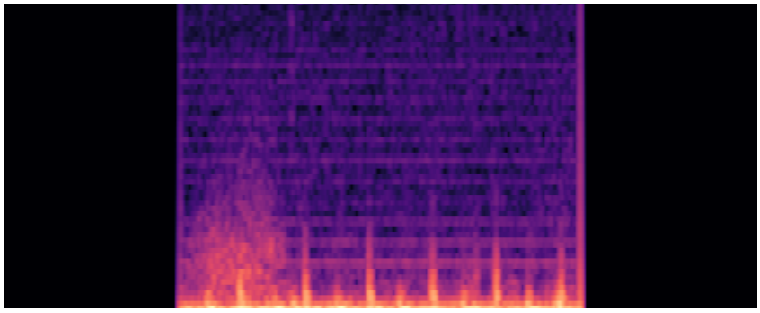


Figura 3.10: Ejemplo de espectrograma correspondiente a un audio.

Un vez tenemos definida la función `convertir_a_spectograma`, debemos escribir el código que se encargue de transformar los audios, que están repartidos en carpetas según su clase y el conjunto de datos al que pertenezca, y guardarlos como imágenes en las correspondientes carpetas. Este extracto de código se puede observar en 3.7.

Código 3.7: Código de generación de los espectrogramas correspondientes a una partición de los ficheros de audio y su archivo como imágenes en las correspondientes carpetas.

```
for s in ['train', 'test', 'val']:
    for cat in ['Healthy','Unhealthy']:
        path_ = path_audios + '/' + s + '/' + cat + '/'
        ficheros = [f for f in os.listdir(path_) if isfile(join(path_, f)) and is_wav(f)]
        for f in ficheros:
          nombre_imagen = path_root+'/output/images/'+s+'/'+cat+'/'+f.split('.wav')[0]+'.png'
          if not os.path.exists(nombre_imagen):
            convertir_a_spectrograma(path_ficheros=path_audios+'/', categoria=cat, fichero=f)
```

Las tres categorías en las que dividimos los datos disponibles son *train* (entrenamiento), *test* y *val* (validación). Un dato a tener en cuenta debido a su gran importancia es que no debemos repartir audios correspondientes a un mismo paciente entre diferentes clases. Esto podría crear un sesgo que no podemos permitir en nuestro algoritmo de aprendizaje. Por tanto, no nos basta con repartir de forma aleatoria los datos en una cierta proporción. Debemos calcular una partición que agrupe los audios de un mismo paciente en la misma categoría.

Para ello, de repartimos los ficheros en las tres clases de forma que, si un paciente ya ha sido adjudicado a una categoría, todos los demás audios correspondientes a este irán a la misma carpeta. Iteramos esta forma de particionar los datos hasta que el error es menor que una cierta tolerancia. Este algoritmo se presenta en el código 3.8. Las proporciones escogidas para la partición son:

- Un $30\,\%$ de los datos irá a la clase de *test*.
- El $70\,\%$ restante se repartirá de la forma

- Un $30\,\%$ (un $21\,\%$ del total) corresponderá a la clase de encargada de validación (*val*).
- El $70\,\%$ restante (un $49\,\%$ del total), a la clase de entrenamiento (*train*).

La función descrita en el código 3.8 se itera hasta que la proporción de audios asignados a cada categoría dentro de cada clase difiere menos de una tolerancia escogida, siendo lo menor posible para que el código terminase de ejecutarse. El valor final para ella es de 5. Posteriormente, simplemente copiamos los ficheros a las carpetas a las que pertenecen utilizando la función `copyfile` de la librería `shutil`.

Código 3.8: Código de reparto de los audios agupando los del mismo paciente en una categoría.

```
for diag in ['Healthy', 'Unhealthy']:
    ficheros = [s.split('.')[0] for s in os.listdir(path = audio_slices_path+'/'+diag)]
    for f in ficheros:
        id_paciente = f.split('_')[0]
        repartido_previamente = False
        for s in ['train', 'test', 'val']:
            if id_paciente in reparto_pacientes[diag][s]:
                repartido_previamente = True
                audios_asignados[diag][s] += 1
        if not repartido:
            if random.random() < 0.3:
                reparto_pacientes[diag]['test'].append(id_paciente)
                audios_asignados[diag]['test'] = 1
            elif random.random() < 0.7:
                reparto_pacientes[diag]['train'].append(id_paciente)
                audios_asignados[diag]['train'] = 1
            else:
                reparto_pacientes[diag]['val'].append(id_paciente)
                audios_asignados[diag]['val'] = 1
```

En la sección 4.2 se explican los cambios que hacemos sobre la generación de los datos para nuestro algoritmo de aprendizaje.

A continuación, teniendo en cuenta lo expuesto en el capítulo 2, debemos diseñar la red neuronal que ejecute el proceso de aprendizaje sobre nuestro conjunto de datos. Como hemos repasado en esa sección, las CNNs tienen amplias aplicaciones en este campo, por lo que emplearemos una de ellas. Además, basaremos el código en el disponible mediante la librería `Keras`, dependiente de `TensorFlow`, una de las más conocidas y utilizadas en aplicaciones de aprendizaje automático. Posteriormente, en la sección 4.3 enunciaremos qué modificaciones se realizan sobre esta red en cada iteración del proyecto.

Para poder analizar el comportamiento de nuestro modelo, hemos de decidir por último qué métricas de las disponibles queremos utilizar para representar y comparar resultados. De las que hemos definido en el capítulo 2, hemos decidido utilizar las conocidas como *Accuracy*, *Precision*, *Recall* y *AUC*.

4

Experimentos y resultados

4.1. Primera aproximación

La primera aproximación a la resolución nuestro problema pasa por utilizar las imágenes obtenidas de los audios disponibles, como se explica en las secciones 3.2 y 3.3. Además, se emplea una CNN ya diseñada y entrenada, en concreto el modelo `VGG16`, con una inicialización de los pesos según la red preentrenada `ImageNet` [14]. El código correspondiente a esta inicialización se puede analizar en 4.1. Además, es necesario, al estar utilizando imágenes como datos de entrada, utilizar el método `ImageDataGenerator().flow_from_directory` disponible por la librería `Keras`, como se puede observar en el código 4.2.

Código 4.1: Código de creación del modelo.

```
vgg16 = VGG16(weights='imagenet')
x = vgg16.get_layer('fc2').output
prediction = Dense(2, activation='sigmoid', name='predictions')(x)
model = Model(inputs=vgg16.input, outputs=prediction)
```

Código 4.2: Código de lectura de las imágenes usando el método `flow_from_directory`.

```
path_train = path_root + '/output/images_meditron_tres_clases/train'
path_test = path_root + '/output/images_meditron_tres_clases/test'
path_val = path_root + '/output/images_meditron_tres_clases/val'

train_data = ImageDataGenerator().flow_from_directory(directory=path_train, target_size=(224,224))
test_data = ImageDataGenerator().flow_from_directory(directory=path_test, target_size=(224,224))
val_data = ImageDataGenerator().flow_from_directory(directory=path_val, target_size=(224,224))
```

Una vez creado el modelo, es necesario especificar una función de *loss*, así como un optimizador y las métricas a utilizar, para poder compilarlo, como se muestra en el código 4.3. Posteriormente, el modelo ya se puede entrenar, llamando al método `model.fit` como se puede observar en el código 4.4. Al ir ejecutando cada época de entrenamiento dentro de un bucle, podemos representar

las gráficas de entrenamiento en tiempo real, para decidir cuándo detener la ejecución. Estas gráficas, tras ejecutar un total de 27 épocas, se pueden observar en la figura 4.1.

Código 4.3: Código de compilación del modelo.

```
opt = Adam(lr=0.01)
model.compile(optimizer=opt, loss=categorical_crossentropy,
              metrics=['accuracy',
                       metrics.Precision(), metrics.Recall(),
                       metrics.AUC()])
```

Código 4.4: Código de ejecución del método `fit`.

```
for e in range(epochs):
  history = model.fit(train_data, steps_per_epoch=train_data.samples//train_data.batch_size,
                      validation_data=val_data,
                      validation_steps=val_data.samples//val_data.batch_size,
                      epochs=1
                      )
```

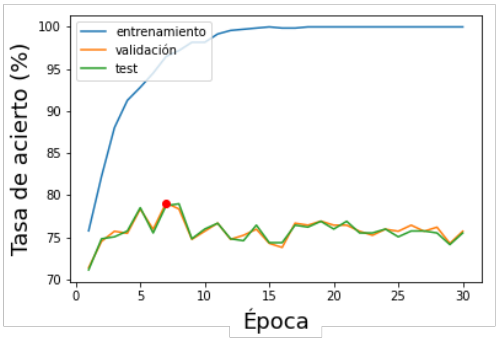
Podemos observar en primer lugar que los datos correspondientes al conjunto de entrenamiento mejoran de forma constante, lo cual es completamente lógico al ser los datos con los que el algoritmo aprende y, por lo tanto, busca minimizar las diferencias. Por otra parte, aunque datos como la precisión o la tasa de acierto mejoran de forma constante para el resto de datos, el valor de la pérdida (*loss*) aumenta en el conjunto de *test* y en el de validación a partir de la época número 5, lo que denota un peor funcionamiento al perder capacidad de generalización. Por lo tanto, en este caso habría que haber terminado la ejecución en una época cercana a esta.

En la tabla 4.1 podemos analizar los valores de la tasa de acierto sobre el conjunto de validación al modificar parámetros, obtenidos mediante el método `model.evaluate` sobre el conjunto de *test*.

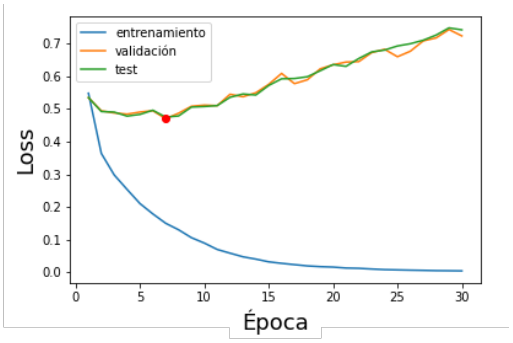
optimizador	*accuracy*	*loss*	*precision*	*recall*	*AUC*
Adam(0.01)	0.7758	24.8813	0.7758	0.7938	0.7552
Adam(0.001)	0.7758	0.5652	0.8170	0.4448	0.7960
Adam(0.0001)	0.8114	0.4456	0.8007	0.8292	0.9160
SGD(0.001)	0.7972	0.4422	0.7966	0.8221	0.9122

Tabla 4.1: Resultados del entrenamiento con varios parámetros.

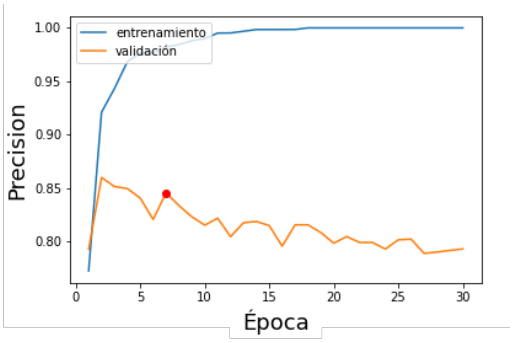
Estos datos parecen mostrar una calidad suficiente teniendo en cuenta, por ejemplo, una tasa de acierto de en torno al 80 %. Sin embargo, hay que tener en cuenta que hay un sesgo entre el número de datos en cada clase. El $77{,}8\,\%$ de los datos corresponden a la clase *Unhealthy*, luego la mejora respecto al modelo que clasifica todos los datos con esta etiqueta no es significativa. Sin embargo,



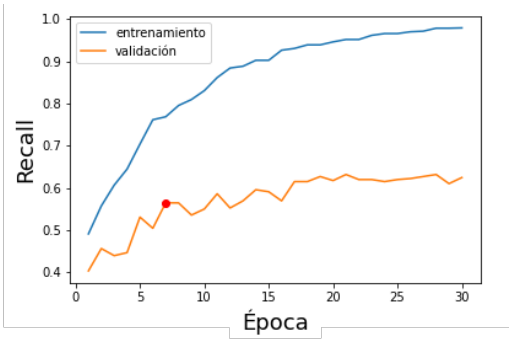
(a) Evolución de la métrica *accuracy*.



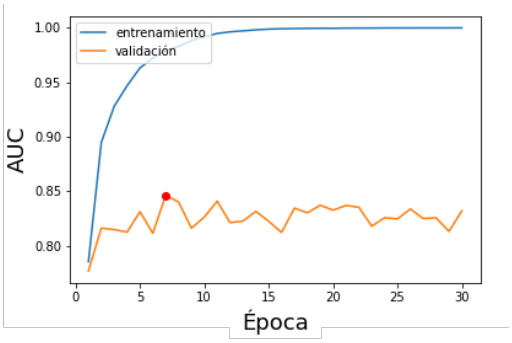
(b) Evolución de la métrica *loss*.



(c) Evolución de la métrica *precision*.



(d) Evolución de la métrica *recall*.



(e) Evolución de la métrica *AUC*.

Figura 4.1: Evolución de las métricas durante el entrenamiento.

este diseño sirve como punto de partida para introducir posteriores modificaciones.

4.2. Modificaciones sobre el procesamiento de los datos

La principal modificación sobre el procesamiento de datos que implementamos es no guardar cada audio en imagen antes de ejecutar el algoritmo de aprendizaje. Entre las razones para dicha modificación se encuentra el usar datos que no estén guardados en disco y sí en *RAM*, lo cual puede acelerar la ejecución. Sin embargo, la principal razón es la pérdida de información que conlleva. El espectrograma que obtenemos de la función `lb.feature.melspectrogram` está almacenado como una matriz de datos con precisión de coma flotante. Sin embargo, al almacenarlo como una imagen, debido a las limitaciones de las mismas, cada valor, que ahora pasa a ser un pixel, tiene un valor de intensidad entero, según el formato `RGB`.

Por tanto, la función `convertir_a_spectrograma`, que antes expusimos en el código 3.6, pasa a ser la función descrita en 4.5. Posteriormente, debemos ejecutar un bucle sobre todos los datos para convertir los audios en datos con los que trabajar, como se puede observar en el código 4.6. Además, hacemos cierto procesamiento sobre estas listas para guardarlas como ficheros *pickle* y no tener que generarlas cada ejecución.

Tras esto, tras una permutación de los datos de cada categoría, escogemos aleatoriamente para tener un reparto equitativo en cada conjunto de los datos de cada una de las dos clases, lo que nos permite comparar nuestro modelo con el que clasifica aleatoriamente (que obtendría una tasa de acierto teórica del $50\,\%$). Por último, normalizamos los datos y los transformamos para definir tensores con los que trabajar nuestra red neuronal, como se expone en el código 4.7.

Código 4.5: Código de generación de los espectrogramas sin perder precisión.

```
def convertir_a_spectrograma(path_ficheros, fichero, categoria, cat='train', verbose=False, mel=True):
    path_completa = path_ficheros + cat + '/' + categoria + '/' + fichero

    y, sr = lb.load(path_completa)
    mel_spect = lb.feature.melspectrogram(y=y, sr=sr, n_fft=2048, hop_length=256)
    mel_spect = lb.power_to_db(mel_spect, ref=np.max)
    src_db=mel_spect/100.0

    return src_db
```

Estas modificaciones cambian la forma de obtener las métricas de entrenamiento, puesto que ahora ya no hay un sesgo dentro de cada categoría hacia una clase, sino que se reparten de forma equitativa en cada una de ellas.

Código 4.6: Código de guardado de los datos en ficheros *pickle*.

```
for s in ['train', 'test', 'val']:
  if os.path.isfile(path_root+'/output/mel/'+s+'.pickle'):
    continue
  data = {'Healthy':[], 'Unhealthy':[]}
  for cat in categorias:
    ficheros = [f for f in os.listdir(path_audios + '/' + s + '/' + cat + '/') if isfile(join(path_audios + '/'
        + s + '/' + cat + '/', f)) and is_wav(f)]
    for f in ficheros:
      mel_data = convertir_a_spectrograma(path_ficheros=path_audios+'/', categoria=cat, fichero=f,
          cat=s, verbose=False, mel=True)
      data[cat].append(mel_data)
  pickle.dump(data, open(path_root+'/output/mel/'+s+'.pickle', 'wb'))

# Cargar los datos de los ficheros binarios
train_data_ = pickle.load(open(path_root+'/output/mel/train.pickle', 'rb'))
test_data_ = pickle.load(open(path_root+'/output/mel/test.pickle', 'rb'))
val_data_ = pickle.load(open(path_root+'/output/mel/val.pickle', 'rb'))
train_data_['Healthy'] = np.array(train_data_['Healthy'])
train_data_['Unhealthy'] = np.array(train_data_['Unhealthy'])
test_data_['Healthy'] = np.array(test_data_['Healthy'])
test_data_['Unhealthy'] = np.array(test_data_['Unhealthy'])
val_data_['Healthy'] = np.array(val_data_['Healthy'])
val_data_['Unhealthy'] = np.array(val_data_['Unhealthy'])
```

4.3. Modificaciones sobre la red de aprendizaje

La primera modificación que realizamos sobre nuestra CNN consiste en diseñar una red desde cero, es decir, no utilizar un modelo como `VGG16` que está diseñado y entrenado para detectar patrones diferentes en las imágenes. Por ello, como primera aproximación definimos el modelo que se describe en 4.8.

Este nuevo modelo tiene una evolución diferente de las métricas respecto a lo analizado anteriormente. Estas se pueden observar en la figura 4.2. Podemos observar que, entre la época 20 y 30, el funcionamiento para los conjuntos diferentes al de entrenamiento empiezan a empeorar lentamente. Para el conjunto de *test*, que no es utilizado para el aprendizaje, obtenemos una tasa de aprendizaje del $58{,}96\,\%$, una precisión de $0{,}6939$ y el valor menos satisfactorio es un *recall* de únicamente $0{,}3277$. Esto implica que existe una cantidad significativa de datos que son etiquetados erróneamente como negativos.

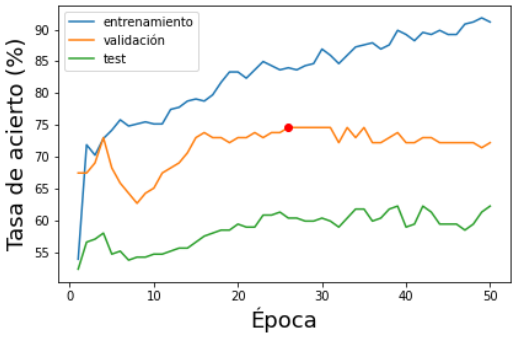
A continuación, la siguiente modificación de la red neuronal consiste en añadir varias capas que añaden funcionalidad, de la forma descrita en la sección 2.2. Estas son `BatchNormalization`, `Dropout` y `GlobalAveragePooling2D`. Además, el número de filtros de las capas `Conv2D` pasa a ser 8 y el tamaño del *kernel* se modifica a ser $(5, 5)$. Se puede observar el código que genera este modelo en 4.9.

Código 4.7: Código de transformación de los datos modificando las dimensiones y normalizando.

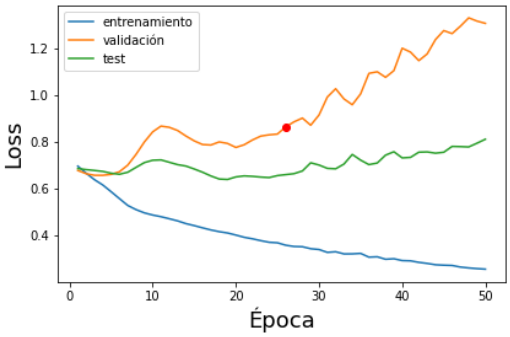
```
ratio = 1.0
np.random.shuffle(train_data_['Unhealthy'])
train_data_['Unhealthy'] = train_data_['Unhealthy'][:int(ratio*len(train_data_['Healthy']))]
np.random.shuffle(val_data_['Unhealthy'])
val_data_['Unhealthy'] = val_data_['Unhealthy'][:int(ratio*len(val_data_['Healthy']))]
np.random.shuffle(test_data_['Unhealthy'])
test_data_['Unhealthy'] = test_data_['Unhealthy'][:int(ratio*len(test_data_['Healthy']))]

X_train = np.concatenate((train_data_['Healthy'], train_data_['Unhealthy']))
X_val = np.concatenate((val_data_['Healthy'], val_data_['Unhealthy']))
X_test = np.concatenate((test_data_['Healthy'], test_data_['Unhealthy']))
y_train = np.array(len(train_data_['Healthy'])*[0] + len(train_data_['Unhealthy'])*[1])
y_val = np.array(len(val_data_['Healthy'])*[0] + len(val_data_['Unhealthy'])*[1])
y_test = np.array(len(test_data_['Healthy'])*[0] + len(test_data_['Unhealthy'])*[1])

inds = np.random.permutation(len(X_train))
X_train = X_train[inds]
y_train = y_train[inds]
inds = np.random.permutation(len(X_val))
X_val = X_val[inds]
y_val = y_val[inds]
inds = np.random.permutation(len(X_test))
X_test = X_test[inds]
y_test = y_test[inds]

X_train = X_train.reshape(*X_train.shape, 1)
X_test = X_test.reshape(*X_test.shape, 1)
X_val = X_val.reshape(*X_val.shape, 1)

def norm(X):
    for i in range(len(X)):
        X[i] = X[i] + 0.8
        flags = ~((X[i,:,:,0])==0).all(axis=0)
        X[i,:,flags] = X[i,:,flags] / X[i,:,flags].mean()
norm(X_train)
norm(X_val)
norm(X_test)
```

Código 4.8: Código para el diseño de un nuevo modelo de CNN

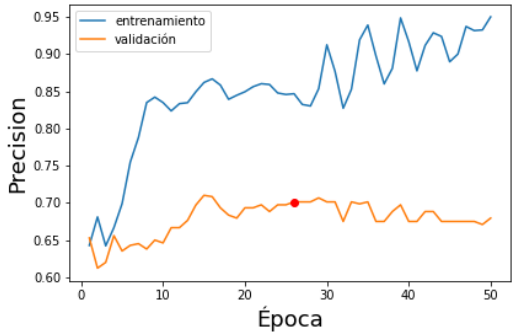
```
model = Sequential()
model.add(Conv2D(8, kernel_size=(4, 2), activation='relu', input_shape=X.shape[1:]))
model.add(MaxPool2D(pool_size=4))
model.add(Conv2D(8, kernel_size=(2, 2), activation='relu'))
model.add(MaxPool2D(pool_size=2))
model.add(Conv2D(8, kernel_size=(2, 1), activation='relu'))
model.add(MaxPool2D(pool_size=2))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
```



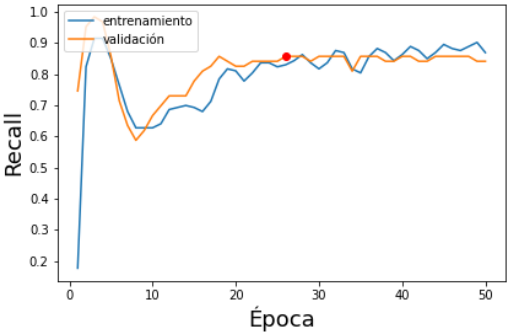
(a) Evolución de la métrica *accuracy*.



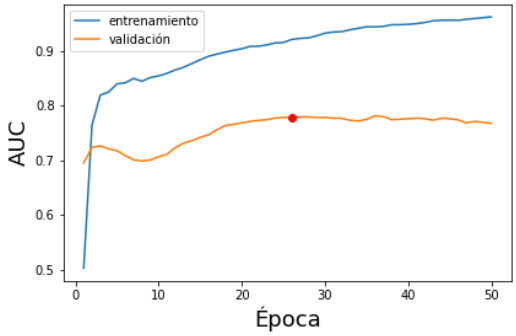
(b) Evolución de la métrica *loss*.



(c) Evolución de la métrica *precision*.



(d) Evolución de la métrica *recall*.



(e) Evolución de la métrica *AUC*.

Figura 4.2: Evolución de las métricas durante el entrenamiento.

Código 4.9: Código para la modificación del modelo de CNN

```
model = Sequential()
model.add(BatchNormalization(input_shape=X.shape[1:]))
model.add(Dropout(rate=0.1, input_shape=X.shape[1:]))
model.add(Conv2D(filters=10, kernel_size=(5,5), activation='relu', strides=2))
model.add(BatchNormalization(momentum=0.1, epsilon=1e-5))
model.add(Dropout(rate=0.15))
model.add(MaxPool2D(pool_size=2))
model.add(Conv2D(filters=10, kernel_size=(5,5), activation='relu', strides=2))
model.add(BatchNormalization(momentum=0.1, epsilon=1e-5))
model.add(Dropout(rate=0.15))
model.add(MaxPool2D(pool_size=2))
model.add(Conv2D(filters=10, kernel_size=(5,5), activation='relu', strides=2))
model.add(BatchNormalization(momentum=0.1, epsilon=1e-5))
model.add(Dropout(rate=0.15))
model.add(MaxPool2D(pool_size=2))
model.add(GlobalAveragePooling2D())
model.add(Dense(1, activation='sigmoid'))
```

Este nuevo modelo presenta unos datos finales sobre el conjunto de *test* diferentes al anterior. En primer lugar, la tasa de acierto aumenta hasta el $60{,}38\,\%$, que no es una mejora significativa. La precisión crece casi un $10\,\%$ hasta un valor de 0.7895, y el *recall* vuelve a ser el peor resultado, con un valor de $0{,}2830$. Finalmente, el valor de la métrica *AUC* aumenta hasta $0{,}6281$.

Finalmente, el último grupo de modificaciones a nuestro modelo se centra en añadir lo conocido como *kernel_regularizer* a las capas encargadas de las convoluciones. Estos regularizadores se encargan de añadir ciertas penalizaciones al valor de la función de *loss* según el valor del *kernel* de la capa. En la librería `Keras` hay tres tipos disponibles: `l1`, `l2` y `l1_l2`, conforme a si se calcula la penalización respecto al valor absoluto de los valores o respecto a su cuadrado. El código 4.10 describe esta inicialización.

Además, aplicamos una normalización diferente a los datos previa a entrenar con ellos. En primer lugar, como por los cálculos que se realizan la mayor parte de las entradas de las matrices valen $-0{,}8$, sumamos el inverso de este número a todas para que sean cero. Tras esto, dividimos por la media de los valores que no son cero. En el código 4.11 se puede observar esta funcionalidad.

La evolución del aprendizaje de esta red, que se puede observar en 4.3, es diferente a la correspondiente a las anteriores ejecuciones.

Los mejores resultados de este modelo se han obtenido con el optimizador `l2`, con un valor del parámetro de $0{,}1$. La tasa de acierto vuelve a aumentar de manera poco significativa hasta el $62{,}26\,\%$, casi un $5\,\%$ de mejora con respecto al primer modelo. La precisión se mantiene casi constante con un valor de $0{,}7826$. El *recall*, de nuevo el valor menos deseable de nuestro modelo, llega a ser $0{,}3396$. Por último, el valor de *AUC* obtiene más de un $10\,\%$ de mejoría hasta alcanzar $0{,}7036$. Todos estos valores

Código 4.10: Código para la mejora del modelo de CNN

```
model = Sequential()
model.add(Conv2D(filters=10, kernel_size=(5,5), activation='relu', input_shape=X.shape[1:], strides=2))
model.add(MaxPool2D(pool_size=2))
model.add(Conv2D(filters=10, kernel_size=(5,5), activation='relu', strides=2,
    kernel_regularizer=regularizers.l2(0.1)))
model.add(MaxPool2D(pool_size=2))
model.add(Conv2D(filters=10, kernel_size=(5,5), activation='relu', strides=2,
    kernel_regularizer=regularizers.l2(0.1)))
model.add(MaxPool2D(pool_size=2))
model.add(GlobalAveragePooling2D())
model.add(Dense(1, activation='sigmoid'))
```

Código 4.11: Código para la normalización de los datos

```
def norm(X):
    for i in range(len(X)):
        X[i] = X[i] + 0.8
        flags = ~((X[i,:,:,0])==0).all(axis=0)
        X[i,:,flags] = X[i,:,flags] / X[i,:,flags].mean()
```

son mejores que los obtenidos con el primer modelo.

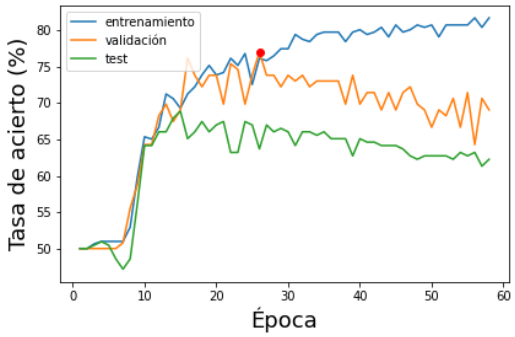
4.4. *Data augmentation*

Por último, realizamos una modificación sin alterar los datos originales ni el modelo, pero que nos permite tener un conjunto de entrenamiento mayor. Como se expone en la sección 2.3, el *data augmentation* es una herramienta útil precisamente para esta finalidad.

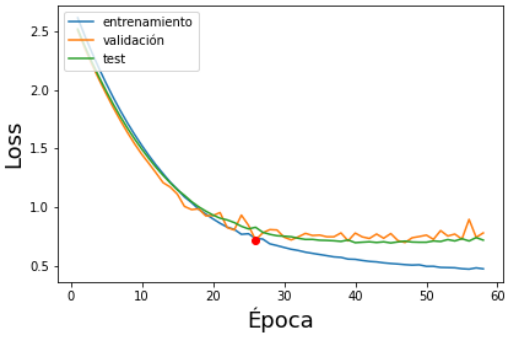
Para ello, en el código 4.12 explicamos cómo implementamos esta funcionalidad. Basándonos en artículos como [32], [33] y [35] decidimos implementar las transformaciones *pitch_shift* y *time_strech*, que modifican el timbre de la grabación y su velocidad, respectivamente, sobre el conjunto de entrenamiento.

En concreto, decidimos aumentar el timbre del audio en un valor medido en semitonos comprendido en el conjunto $\{-3{,}5, -2{,}5, -2, -1, 1, 2, 2{,}5, 3{,}5\}$. Además, la longitud del audio se reescala según una multiplicación por un valor en $\{0{,}8, 0{,}9, 1{,}1, 1{,}2\}$. Este último cambio hace que sea necesario volver a recortar la parte central del audio o añadir más silencio al mismo para mantener la longitud que habíamos escogido de 6 segundos.

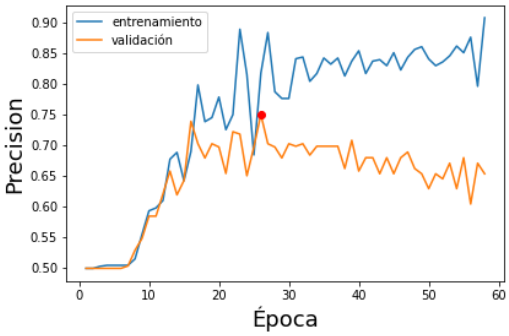
Esta serie de modificaciones hacen que multipliquemos la cantidad de audios de entrenamiento por 13, pasando de un total de 742 a 9646 audios.



(a) Evolución de la métrica *accuracy*.



(b) Evolución de la métrica *loss*.



(c) Evolución de la métrica *precision*.

Figura 4.3: Evolución de las métricas durante el entrenamiento.

Código 4.12: Código para la implementación del *data augmentation*

```
def audio_a_spectrograma(y, sr):
    mel_spect = lb.feature.melspectrogram(y=y, sr=sr, n_fft=2048, hop_length=256)
    return = lb.power_to_db(mel_spect, ref=np.max)/100.0

def convertir_a_spectrograma_augmentation(path_ficheros, fichero, categoria, cat='train'):
    # Original
    y, sr = lb.load(path_completa)
    res.append(audio_a_spectrograma(y, sr))
    # Aplicamos shift
    for shift in [-3.5, -2.5, -2, -1, 1, 2, 2.5, 3.5]:
      y_shift = lb.effects.pitch_shift(y, sr, n_steps=1, bins_per_octave=12)
      res.append(audio_a_spectrograma(y_shift, sr))
    # Aplicamos time stretch
    for stretch in [0.8, 0.9, 1.1, 1.2]:
      y_stretch = lb.effects.time_stretch(y, stretch)
      # Recuperamos longitud objetivo
      sobrante = len(y_stretch) -132300
      if sobrante > 0:
        y_stretch = y_stretch[int(sobrante/2+1):-int(sobrante/2)]
      y_stretch = lb.util.pad_center(y_stretch, 132300)
      res.append(audio_a_spectrograma(y_stretch, sr))
    return res
```

Estos cambios conllevan una alteración de los resultados de nuestro algoritmo de aprendizaje. En concreto, la tasa de acierto aumenta hasta un valor de $0{,}6415$, un $15\,\%$ mayor que el modelo inicial; se obtiene una precisión de un $0{,}7778$, un cambio despreciable respecto al anterior modelo; y el *AUC* se incrementa hasta un valor de $0{,}7374$. El *recall*, que durante todo este trabajo ha sido la métrica con peores resultados, alcanza su máximo de $0{,}3962$, frente al $0{,}3277$ inicial.

Para finalizar este estudio, empleamos otra manera de utilizar el *data augmentation*. Para ello, hacemos uso de la clase `ImageDataGenerator` incluida en la librería `Keras`. Como se puede observar en el código 4.13, mediante ciertos parámetros le indicamos a esta función cómo generar nuevos datos. En concreto, le decimos, mediante el parámetro *width_shift_range*, que desplace los audios en el eje temporal un valor aleatorio menor que el $20\,\%$ de la longitud original, tanto positivo como negativo. Similar es el uso de *height_shift_range*, que desplaza ligeramente, un máximo de un $1\,\%$, el eje vertical de los datos. Por último, el parámetro *brightness_range* altera el brillo de la imagen que representan los datos a un valor entre el $20\,\%$ y el $120\,\%$ del original. Debido a que en este estudio los datos representan audios, en concreto respiraciones, no tiene sentido aplicar otras transformaciones como simetrías o rotaciones de los datos.

Tras esta modificación, la forma de ejecutar el método `model.fit()`, a la que es necesaria especificar los datos disponibles para el entrenamiento, como se puede observar en el código 4.4, pasa a ser el expuesto en 4.14. En este, se llama a una función que invoca al algoritmo de generación de

Código 4.13: Código para una nueva implementación del *data augmentation*

```
datagen = ImageDataGenerator(width_shift_range=0.2,
                             height_shift_range=0.01,
                             brightness_range=[0.2,1.2])
```

datos con modificaciones aleatorias según los parámetros especificados.

Código 4.14: Código de ejecución del método `fit_generator`

```
history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=batch_size),
                              epochs=1, callbacks=[checkpoint], shuffle=True,
                              steps_per_epoch=len(X_train)//batch_size,
                              class_weight=class_weights, validation_data=(X_val,y_val))
```

De nuevo, esta modificación del entrenamiento conlleva un cambio en los resultados del apredizaje. En primer lugar, la tasa de acierto vuelve a aumentar hasta alcanzar un valor de $0{,}6921$. Esto representa un incremento de más de una décima respecto al modelo inicial. La precisión aumenta de manera considerable hasta llegar a ser $0{,}8393$, el valor más alto hasta el momento. La métrica *AUC* experimenta un ligero incremento hasta $0{,}7395$. El *recall* permanece la métrica con peores resultados, aunque repunta hasta un valor de $0{,}4174$.

El conjunto de estos resultados ilustra un mejor funcionamiento que el modelo inicial en todos los aspectos tenidos en cuenta por estas métricas. En concreto, obtenemos una tasa de acierto de prácticamente el $70\,\%$, lo cual es una mejora teniendo en cuenta el reparto equitativo de datos en cada clase.

5

Conclusiones y trabajo futuro

En primer lugar, el algoritmo de preprocesamiento diseñado elimina aquellas partes de los audios que no contienen información relevante para nuestro estudio, separando cada respiración. Al hacer un estudio sobre las duraciones de estas, hemos podido mantener el mayor contenido posible sin resultar en datos que compliquen el diseño de la red neuronal que aprenda sobre ellos. Esto ha permitido, además, que nuestro modelo tenga la posibilidad futura de utilizar datos diferentes a los disponibles originalmente.

Teniendo en cuenta los resultados sobre un conjunto de datos independiente del de entrenamiento, así como métricas que consideran conjuntos como los falsos positivos o falsos negativos, como pueden ser las denominadas *precision*, *recall* y *AUC*, la CNN seleccionada presenta los mejores valores entre todas las diseñadas y analizadas.

Adicionalmente, como el conjunto de datos con el que hemos trabajado ha resultado limitado, hemos empleado *data augmentation*, que ha conseguido mejorar los resultados obtenidos al incrementar la cantidad de datos disponibles para el entrenamiento.

Partiendo de un reparto equitativo de audios de ambas clases en cada conjunto, diseñamos un modelo que obtiene una tasa de acierto del $62{,}26\,\%$. Tras aplicar el *data augmentation* y aumentar la cantidad de datos disponibles, este valor se eleva hasta el $69{,}21\,\%$. Este aumento representa una mejoría considerable, de casi un $10\,\%$. Esto implica que nuestro modelo podría obtener mejores resultados, si se dispusiese de una mayor cantidad de datos.

El modelo diseñado presenta unas métricas de comportamiento que distan de ser óptimas, pero permite ser un punto de partida para diseñar nuevos algoritmos que tengan en cuenta la problemática de este tipo de estudios.

Como trabajo futuro, sería interesante poder probar este modelo con un conjunto de datos mayor al disponible para este estudio. Nuestra red, diseñada como una CNN, tiene potencial para aprender de ellos y de poder obtener mejores resultados.

Por otra parte, sería también interesante incluir métodos adicionales de *data augmentation*, para que el modelo pueda aprender aún más a no tener en cuenta factores innecesarios para la clasificación,

como el timbre de la voz o el ruido de fondo, que puedan existir en la grabación.

Finalmente, otro posible enfoque de trabajo futuro a desarrollar sobre este estudio consiste en utilizar otro tipo de redes neuronales para la misma finalidad. Sería interesante escoger modelos diseñados para el análisis de series temporales, ya que podemos tomar el audio como la evolución a lo largo del tiempo de datos como el timbre o frecuencias. Un ejemplo sería, por tanto, aplicar las conocidas como *Long Short-Term Memory* (LSTM) , o sus derivaciones, denominadas *Gated Recurrent Unit* (GRU) . Estas son ejemplos de redes neuronales recurrentes, que analizan la evolución de sus entradas respecto al tiempo. Estas redes tienen amplias aplicaciones, entre otras, en reconocimiento de patrones de voz, muy relacionado con la temática de este trabajo.

Bibliografía

[1] B. M. Rocha, D. Filos, L. Mendes, I. Vogiatzis, E. Perantoni, E. Kaimakamis, P. Natsiavas, A. Oliveira, C. Jácome, A. Marques, R. P. Paiva, I. Chouvarda, P. Carvalho, and N. Maglaveras, "A Respiratory Sound Database for the Development of Automated Classification," in *Precision Medicine Powered by pHealth and Connected Health* (N. Maglaveras, I. Chouvarda, and P. de Carvalho, eds.), (Singapore), pp. 33–37, Springer Singapore, 2018. (Disponible para descargar).

[2] N. S. Punn, S. K. Sonbhadra, and S. Agarwal, "Covid-19 epidemic analysis using machine learning and deep learning algorithms," *medRxiv*, 2020.

[3] S. F. Ardabili, A. Mosavi, P. Ghamisi, F. Ferdinand, A. R. Varkonyi-Koczy, U. Reuter, T. Rabczuk, and P. M. Atkinson, "Covid-19 outbreak prediction with machine learning," *Algorithms*, vol. 13, no. 10, 2020.

[4] A. M. Ismael and A. Şengür, "Deep learning approaches for covid-19 detection based on chest x-ray images," *Expert Systems with Applications*, vol. 164, p. 114054, 2021.

[5] G. Rong, A. Mendez, E. Bou Assi, B. Zhao, and M. Sawan, "Artificial Intelligence in Healthcare: Review and Prediction Case Studies," *Engineering*, vol. 6, no. 3, pp. 291–301, 2020.

[6] K. Sharma, A. Kaur, and S. Gujral, "Brain tumor detection based on machine learning algorithms," *International Journal of Computer Applications*, vol. 103, no. 1, pp. 7–11, 2014.

[7] Z. Wang, G. Yu, Y. Kang, Y. Zhao, and Q. Qu, "Breast tumor detection in digital mammography based on extreme learning machine," *Neurocomputing*, vol. 128, pp. 175–184, 2014.

[8] N. Howlader, A. Noone, M. Krapcho, D. Miller, A. Brest, M. Yu, J. Ruhl, Z. Tatalovich, A. Mariotto, D. Lewis, H. Chen, E. Feuer, and K. Cronin, "SEER Cancer Statistics Review, 1975-2018," (Disponible).

[9] C. Acebes, X. Morales, and O. Camara, "A Cartesian Grid Representation of Left Atrial Appendages for a Deep Learning Estimation of Thrombogenic Risk Predictors," in *Statistical Atlases and Computational Models of the Heart. M&Ms and EMIDEC Challenges* (E. Puyol Anton, M. Pop, M. Sermesant, V. Campello, A. Lalande, K. Lekadir, A. Suinesiaputra, O. Camara, and A. Young, eds.), (Cham), pp. 35–43, Springer International Publishing, 2021.

[10] T. J. Brinker, A. Hekler, A. H. Enk, J. Klode, A. Hauschild, C. Berking, B. Schilling, S. Haferkamp, D. Schadendorf, T. Holland-Letz, J. S. Utikal, and C. von Kalle, "Deep learning outperformed 136 of 157 dermatologists in a head-to-head dermoscopic melanoma image classification task," *European Journal of Cancer*, vol. 113, pp. 47–54, May 2019.

[11] A. Buetti-Dinh, V. Galli, S. Bellenberg, O. Ilie, M. Herold, S. Christel, M. Boretska, I. V. Pivkin, P. Wilmes, W. Sand, M. Vera, and M. Dopson, "Deep neural networks outperform human expert's capacity in characterizing bioleaching bacterial biofilm composition," *Biotechnology Reports*, vol. 22, p. e00321, 2019.

[12] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, M. Slaney, R. J. Weiss, and K. Wilson, "CNN architectures for large-scale audio classification," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 131–135, 2017.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[14] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 2015.

[15] M. Lin, Q. Chen, and S. Yan, "Network In Network," 2014.

[16] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, Feb. 2017.

[17] J. Xie, K. Hu, M. Zhu, J. Yu, and Q. Zhu, "Investigation of Different CNN-Based Models for Improved Bird Sound Classification," *IEEE Access*, vol. 7, pp. 175353–175361, 2019.

[18] Y. Yin, R. R. Shah, and R. Zimmermann, "Learning and Fusing Multimodal Deep Features for Acoustic Scene Categorization," in *Proceedings of the 26th ACM International Conference on Multimedia*, MM '18, (New York, NY, USA), p. 1892–1900, Association for Computing Machinery, 2018.

[19] R. Hyder, S. Ghaffarzadegan, Z. Feng, J. H. Hansen, and T. Hasan, "Acoustic Scene Classification Using a CNN-SuperVector System Trained with Auditory and Spectrogram Image Features," in *Proc. Interspeech 2017*, pp. 3073–3077, 2017.

[20] Y. Su, K. Zhang, J. Wang, and K. Madani, "Environment Sound Classification Using a Two-Stream CNN Based on Decision-Level Fusion," *Sensors*, vol. 19, no. 7, 2019.

[21] M. Won, A. Ferraro, D. Bogdanov, and X. Serra, "Evaluation of CNN-based Automatic Music Tagging Models," 2020.

[22] E. Law, K. West, M. I. Mandel, M. Bay, and J. S. Downie, "Evaluation of algorithms using games: The case of music tagging.," in *ISMIR*, pp. 387–392, 2009.

[23] T. Bertin-Mahieux, D. P. W. Ellis, B. Whitman, and P. Lamere, "The Million Song Dataset," 2011.

[24] D. Bogdanov, M. Won, P. Tovstogan, A. Porter, and X. Serra, "The MTG-Jamendo Dataset for Automatic Music Tagging," in *Machine Learning for Music Discovery Workshop, International Conference on Machine Learning (ICML 2019)*, (Long Beach, CA, United States), 2019.

[25] J. Zhao, X. Mao, and L. Chen, "Speech emotion recognition using deep 1D & 2D CNN LSTM networks," *Biomedical Signal Processing and Control*, vol. 47, pp. 312–323, 2019.

[26] F. Burkhardt, A. Paeschke, M. Rolfes, W. F. Sendlmeier, and B. Weiss, "A database of German emotional speech," in *Ninth European Conference on Speech Communication and Technology*, 2005. (Disponible para descargar).

[27] C. Busso, M. Bulut, C.-C. Lee, A. Kazemzadeh, E. Mower, S. Kim, J. N. Chang, S. Lee, and S. S. Narayanan, "IEMOCAP: interactive emotional dyadic motion capture database," *Language*

Resources and Evaluation, vol. 42, pp. 335–359, Nov. 2008.

[28] A. Meghanani, A. C. S., and A. G. Ramakrishnan, "An Exploration of Log-Mel Spectrogram and MFCC Features for Alzheimer's Dementia Recognition from Spontaneous Speech," in *2021 IEEE Spoken Language Technology Workshop (SLT)*, pp. 670–677, 2021.

[29] S. Luz, F. Haider, S. de la Fuente, D. Fromm, and B. MacWhinney, "Alzheimer's Dementia Recognition through Spontaneous Speech: The ADReSS Challenge," in *Proceedings of INTERSPEECH 2020*, (Shanghai, China), 2020.

[30] H. Meng, T. Yan, F. Yuan, and H. Wei, "Speech Emotion Recognition From 3D Log-Mel Spectrograms With Deep Learning Network," *IEEE Access*, vol. 7, pp. 125868–125881, 2019.

[31] M. Sahidullah and G. Saha, "Design, analysis and experimental evaluation of block based transformation in MFCC computation for speaker recognition," *Speech Communication*, vol. 54, no. 4, pp. 543–565, 2012.

[32] J. Salamon and J. P. Bello, "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification," *IEEE Signal Processing Letters*, vol. 24, no. 3, pp. 279–283, 2017.

[33] N. Davis and K. Suresh, "Environmental Sound Classification Using Deep Convolutional Neural Networks and Data Augmentation," in *2018 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pp. 41–45, 2018.

[34] B. McFee, E. J. Humphrey, and J. P. Bello, "A software framework for musical data augmentation.," in *ISMIR*, vol. 2015, pp. 248–254, 2015.

[35] Z. Mushtaq and S.-F. Su, "Environmental sound classification using a regularized deep convolutional neural network with data augmentation," *Applied Acoustics*, vol. 167, p. 107389, 2020.

[36] T. Inoue, P. Vinayavekhin, S. Wang, D. Wood, A. Munawar, B. J. Ko, N. Greco, and R. Tachibana, "Shuffling and mixing data augmentation for environmental sound classification," 2019.

Acrónimos

BMI Índice de masa corporal.

CNN Red Neuronal Convolucional.

COPD *Chronic Obstructive Pulmonary Disease*.

DCT Transformada de coseno discreta.

ELM *Extreme Learning Machine*.

GRU *Gated Recurrent Unit*.

LRTI *Lower Respiratory Tract Infection*.

LSTM *Long Short-Term Memory*.

STFT Transformada de Fourier de Tiempo Reducido.

SVM *Support Vector Machine*.

URTI *Upper Respiratory Tract Infection*.